MASTER EN SCIENCES INFORMATIQUES

ORKA obstacles recognition with KAOS based on ASAX
utilisation d'ASAX pour la détection d'obstacle

Brohez, Simon; Grégoire, Yann

Award date:
2002

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'Informatique

ORKA – Obstacles Recognition with KAOS based on ASAX

-

Utilisation d'ASAX pour la détection d'obstacle

-

Simon Brohez & Yann Grégoire

Année Académique 2001 – 2002

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique

Résumé

Le monitoring dynamique de programme est une approche qui peut être utilisée dans le cadre d'une analyse des requirements, pour vérifier si l'implémentation d'une solution logicielle correspond à son comportement attendu. Nous présentons ORKA, un outil développé pour monitorer l'implémentation d'un système spécifié avec la méthode d'analyse des requirements, orientée buts, KAOS. Cet outil traduit les spécifications KAOS en règles ASAX. ASAX est un outil efficace d'analyse de fichiers séquentiels, utilisé auparavant principalement en détection d'intrusions. L'approche KAOS utilise un sous-ensemble de la logique temporelle (un ensemble de patterns prédéfinis). Les formules KAOS sont traduites en règles ASAX utilisées pour analyser une trace d'exécution du système. Nous comparerons également notre approche à d'autres existantes, pour en juger la pertinence et l'efficacité. Cette analyse permet de déterminer si la spécification est (in)complète et de la revoir le cas échéant. Nous développons finalement quelques jeux de ests et comparerons également notre approche à d'autres existantes, pour en juger la pertinence et l'efficacité.

Abstract

Dynamic program monitoring is an approach that can be used in requirement analysis to check whether an implementation of a solution corresponds to the expected behaviour of the program. We present ORKA, a tool developed to monitor an implementation of a system specified with KAOS, a goal oriented requirement analysis method. This tool translates KAOS specifications to ASAX rules. ASAX is an efficient sequential file analysis tool, previously used mainly for intrusion detection. The KAOS approach uses a subset of Temporal Logic (a set of predefined patterns). KAOS formulas are translated to ASAX rules that are used by ASAX to monitor a system trace. The expressiveness and efficiency of our approach is compared to other similar systems. This analysis allow us to determine if the specification is (in)complete and to re-examine it if necessary. Finally, we make of couple of tests and compare our existing approach with others, to judge its relevance and its efficiency.

Remerciements

Au Professeur Baudouin Le Charlier,
pour la qualité de ses conseils, sa disponibilité, son écoute et sa patience ;

Au Professeur Axel van Lamsweerde,
pour la connaissance de KAOS qu'il nous a transmise ;

A Nicolas, Xavier, Vincent et Christophe,
pour leur encadrement et leur expertise, tant pour KAOS que pour ASAX,

Aux Isabelle(s), Loïc, Raphaël, Christophe, Valentin,
Emmanuel, Marie-France, Francis, Gustavo, Freddy, Etienne… et tous les autres
qui nous ont chaleureusement accueilli au sein du département Informatique de l'UCL ;

A nos amis et camarades de classe,
qui nous ont fait vivre de superbes années au sein l'environnement universitaire ;

Et, enfin, l'un à l'autre,
pour la confiance, la bonne humeur et la motivation
qui ont régnées tout au long de ce travail.

Table des Matières

Table des Figures

1. INTRODUCTION

Les développements actuels de l'informatique en général et de la programmation en particulier ne vont pas sans poser un certain nombre de problème au niveau du contrôle, de qualité, de correction, d'efficacité… Dans un environnement de plus en plus complexe, au niveau de ses techniques et de ses méthodes, où les délais sont devenus de plus en plus contraignants et où le temps des divers intervenants humains est devenu la ressource critique, et donc la plus chère, il est nécessaire de fournir des outils et méthodes pouvant procurer une « aide au contrôle ».

Que ce soit au niveau du programmeur, se trouvant face à un problème à spécifier et implémenter correctement, que ce soit au niveau d'un chef de projet, devant jongler avec différents objectifs à implémenter dans un même système dont il a la responsabilité, que ce soit, enfin, au niveau du décideur, qui veut *effectivement* obtenir ce qu'il a demandé, tous ont intérêt à se munir d'outils permettant d'appréhender la complexité actuelle des programmes sans devoir pour autant dépenser leur temps, voire leur argent, dans de longues et fastidieuses analyses.

Bien entendu, chacun de ces acteurs ne sera pas intéressé par tous les types de contrôle possibles. Le programmeur voudra des spécifications correctes, sans failles, correspondant aux fonctionnalités définies sur le système. Il a à sa disposition des outils de « model checking », des automates, des formalismes divers et variés.

Le chef de projet, quant à lui, désire modéliser un système. Ce système doit être entièrement formalisé, dans un langage qui puisse satisfaire le client (par sa clarté) et le programmeur (par sa précision). Diverses contraintes doivent être prises en compte, certaines purement techniques, d'autres économiques, légales… KAOS[1] est l'un de ces systèmes. Il permet de répondre aux attentes en terme de modélisation de système, en permettant d'obtenir une vue globale des différents objectifs, des contraintes, des relations entre les composants du système, de l'évolution du système dans le temps…

Le décideur, le commanditaire, lui, n'a que faire des outils de « model checking ». Il n'est pas non plus intéressé par les spécifications des différents composants, et n'ira pas « lire le code » pour s'assurer que telle ou telle fonctionnalité est correctement implémentée. Bien entendu, il a négocié certains objectifs, et un modèle, de type KAOS, peut être utile dans la relation avec le client. Mais ce n'est plus la qualité des *spécifications* qui nous intéresse ici, c'est la qualité de *l'implémentation*. Et pour cela, les outils mis à la disposition des différents acteurs ne sont pas légion.

Que faire, à ce niveau, sinon une batterie de tests ? Et comment interpréter ces tests ? Comment trouver l'erreur, si erreur il y a ? La réponse est simple, du moins dans ses principes : puisque nous avons modélisé le comportement du système, nous pouvons également le surveiller, le « monitorer ».

[1] [VLDDD91]

C'est ici qu'intervient ASAX[2], qui est un logiciel d'audit. Il aura pour mission de vérifier le comportement du système, à l'exécution, de manière dynamique, en situation d'utilisation réelle. Il pourra aussi (et surtout !) renseigner son utilisateur sur le (les) objectifs posant problème, dire où, quand et pourquoi certaines séquences d'utilisation mettent en défaut le système, et ceci de façon tout à fait systématique (et donc reproductible).

Qu'est-ce donc que ORKA ? Une interface, un logiciel de traduction, permettant de définir, à partir du modèle KAOS d'un système, ses règles de fonctionnement en RUSSEL, qui est le langage utilisé au sein d'ASAX. Les contraintes définies dans KAOS devenant des règles RUSSEL, directement utilisables par ASAX.

L'idée est donc de partir d'un modèle KAOS existant (l'utilisateur d'ORKA ne se substituant *pas* à l'utilisateur de KAOS qui reste le responsable de la modélisation du système) et d'en dériver les buts à « monitorer ». Ces buts, définis en logique temporelle, sont alors traduits sous forme de règles, et ces règles sont appliquées aux traces d'exécution du programme.

Nous vous présenterons donc, pour commencer, KAOS, ses tenants et aboutissants, son méta modèle, la façon dont on spécifie les buts, les agents, les contraintes… et bien entendu un exemple pour illustrer le tout.

Ensuite, nous ferons de même pour ASAX, en insistant sur son mode de fonctionnement, ses concepts (principalement les règles).

La sémantique étant le problème crucial de toute traduction, nous vous proposerons une analyse de la sémantique d'un sous-ensemble de la logique temporelle, utilisée dans KAOS.

Enfin, nous vous présenterons la traduction proprement dite, en utilisant différents formalismes, permettant, étape par étape, de passer de la sémantique des formules de logique temporelle aux règles RUSSEL correspondantes, en passant par des fonctions, puis par des organigrammes. Nous mettrons l'accent à chaque fois sur la nécessité, d'une part, de conserver intacte la sémantique originelle, et, d'autre part, d'intégrer au fur et à mesure les concepts opérationnels induits par l'exécution du programme correspondant au système modélisé.

Le lecteur intéressé poura également consulter l'article soumis pour publication[3] relatif à ce mémoire[4]

[2] [HLCMM92]
[3] RV'02 - Second Workshop on *Runtime Verification* (http://ase.arc.nasa.gov/rv2002/)
[4] www.info.fundp.ac.be/~sbrohez/ORKA

2. KAOS

2.1. Présentation

La conception d'un logiciel[5] passe par une première phase où les concepteurs et demandeurs du produit se rencontrent. Cette première phase est primordiale car elle va être le départ de la collaboration entre les différentes parties en rapport avec le logiciel: clients, utilisateurs, concepteurs, développeurs. Elle est appelée, en anglais, requirement analysis, ou analyse des besoins (on dit aussi analyse des exigences).

C'est du bon « traitement » de cette phase que va découler un logiciel en adéquation avec ce que les utilisateurs voulaient. Point clef de la conception, son exactitude va déterminer le niveau de qualité du produit développé.

L'analyse des besoins est une phase trop souvent mise à mal par les concepteurs. Elle est souvent bâclée, pour ne pas dire inexistante. Or, une mauvaise analyse entraînera la conception du logiciel vers une mauvaise route. On pourra corriger le tir, mais avec de grandes dépenses en temps, hommes, argent… si le projet n'est pas abandonné.
C'est pourquoi il y a tellement de gens qui se penchent sur ce que l'on appelle maintenant l'ingénierie des exigences (requirement engineering).

L'analyse des besoins, comme on l'a dit, cherche à donner au concepteur une vue du système dans lequel son logiciel va s'exécuter, et également l'ensemble des buts que veulent atteindre les utilisateurs finaux du logiciel. Une foule de choses sont inclues dans ceci: les besoins (on en a déjà parlé), les hypothèses faites (souvent implicitement) sur le système, l'environnement intérieur et extérieur du système, etc.

Plusieurs problèmes apparaissent directement. Cette phase est tout d'abord loin d'être suffisante. Comment peut-on être sûr de ne rien avoir oublié ? Les utilisateurs (demandeurs) n'ont-ils rien oublié ? Les concepteurs ont-ils tout vu ? Une première vision comme celle-là est beaucoup trop idéaliste, trop optimiste. Trop peu de choses sont prises en compte. Conséquence: le logiciel développé risque de ne pas être assez robuste, d'avoir un comportement anormal ou inattendu face à un cas d'utilisation non spécifié.

En fait, cette étape n'est que la première de la collaboration (nécessaire) entre demandeurs et concepteurs. Un suivi de l'évolution de la conception du logiciel devra être effectué, avec moult réunions, afin d'affiner les positions des uns et des autres, pour améliorer la tournure du projet. Ceci afin de garantir l'efficacité, l'efficience et la robustesse du produit développé.

Des solutions ont été apportées bien évidemment. Tout d'abord à travers la collaboration continue entre concepteurs et demandeurs du produit, et également à

[5] Cours de « Génie Logiciel », 2ème Maîtrise (http://www.info.fundp.ac.be/~nha/plangl.htm) et cours de « Méthodologie de Développement de Logiciels: matières approfondies », 3ème maîtrise (http://www.info.fundp.ac.be/~software-quality/students/), tous deux enseignés par Mr. N. Habra.

travers d'autres techniques/méthodes de travail. Plusieurs langages de spécification, comme le langage Z [6], ont été développés afin mieux élaborer l'étape d'analyse des besoins. Ces langages n'agissent pas dans les mêmes catégories de paradigmes: on trouve des spécifications d'états, d'historique, opérationnelles, algébriques, ou encore basées sur les transitions du système.

Il y a cependant des problèmes dans l'utilisation de ces solutions. Tout d'abord au point de vue de la modélisation des exigences. Les objets, transitions et opérations sont définis de manière semi formelle, les modèles étant trop tournés vers les aspects fonctionnels, le non fonctionnel (exploitabilité, fiabilité, performance, maintenabilité, interfaces,etc.) étant souvent lésé. Certaines techniques utilisent des spécifications formelles, mais elles sont difficiles à écrire, et encore plus à comprendre et à communiquer aux clients investis dans le projet. Ensuite, ces techniques se limitent souvent à poser des questions sur « quoi » portera le produit, sans vraiment savoir « pourquoi » il est développé, ni « qui » est vraiment concerné par le produit. De plus, la frontière entre exigences, description du domaine d'application et hypothèses sur le système est généralement très floue, pour ne pas dire inexistante.

Le nom du projet KAOS apparaît pour la première fois dans [DVLF93] en 1993. KAOS, qui signifie Knowledge Acquisition in autOmated Specification, est une méthode d'acquisition des exigences. Cette méthode propose une autre approche de l'acquisition des besoins, une approche orientée vers les buts que le système, aidé du logiciel, devra atteindre. Pourquoi s'orienter vers les buts ? Cette approche propose un modèle plus stable du système, définissant les exigences de manière plus rationnelle, en orientant leur identification dans une optique de satisfaction des buts. De plus, cette technique permet plus facilement de détecter et résoudre les conflits au sein du système. En effet, les buts de chacun des intervenants sont souvent en adéquation (conflits), et difficiles à percevoir. Par cette approche, ils sont directement perçus, et peuvent donc être résolus directement, alors qu'il aurait fallu beaucoup plus de temps pour (éventuellement) les remarquer avec un autre processus de spécification.

2.2. Fonctionnement

L'approche de KAOS utilise des concepts et techniques venant du domaine de l'intelligence artificielle. Les modèles d'exigences trouvés font partie d'un méta-modèle conceptuel. En effet, on décompose ici l'acquisition de ces exigences en plusieurs niveaux, comme le présente la figure 1 ci-dessous. Un niveau « méta » (meta level), qui définit des méta-concepts supportés le langage de spécification (présenté plus loin), leurs méta-attributs, leurs méta-relations, les méta-contraintes associées,…

Un deuxième niveau, le niveau du domaine d'application (domain level), contient les instances de ces méta-concepts et méta-relations qui rentrent dans le domaine du système sur lequel nous travaillons. Par exemple, le concept d'« Exemplaire » est une instance du méta-concept « entité », « Emprunter » est une instance du méta-concept d' « action », et ainsi de suite.

[6] http://www.afm.sbu.ac.uk/z/, d'autres sont cités dans [Dar93]

Le dernier niveau, ou niveau d'instanciation (instance level), est constitué d'instances spécifiques des concepts du niveau du domaine, comme la « TheseAMounji97» qui est une instance du concept de « Document ».

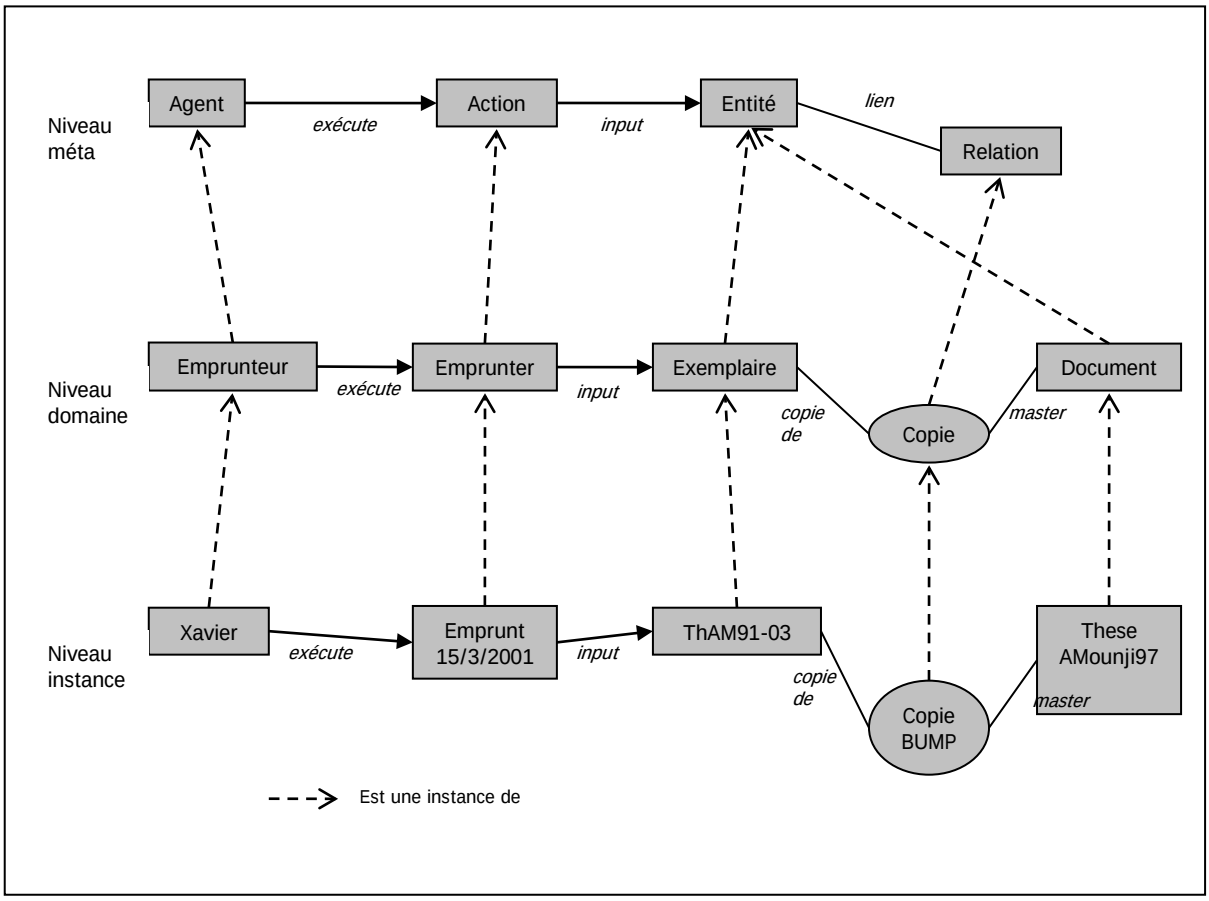


fig. 2.1 - niveaux méta, domaine et instance

2.1.1 Le méta-modèle KAOS

Le méta-modèle est donc un modèle pour le niveau « méta ». Il guidera l'acquisition des concepts au niveau du domaine d'application.
Les composants de ce méta-modèle que nous allons utiliser sont définis ci-dessous.[7]

- *Objet* : Un objet est un composant du système dans les instances peuvent évoluer d'état en état (suite à l'exécution d'une action). Les objets sont en général définis plus précisément en tant qu'*entité*, *relation*, ou *événement*, suivant que l'objet soit respectivement autonome, subordonné, ou instantané. Les objets sont caractérisés par des attributs et des invariants.

- *Action* : Une action est une relation entre des objets, vue comme une opération (mathématique) d'input-output. Une action qui est exécutée correspond à une transition entre deux états du système. Les actions sont

[7] Une version plus précise peut être trouvée dans [DVLF93].

caractérisées par des pré- et post-conditions, mais aussi par des conditions de déclenchement (trigger-condition)[8].

- *Agent* : Un agent est un objet qui exécute une action, à condition qu'il soit associé à cette action. Un agent a accès/peut contrôler un objet s'il peut en voir/contrôler les états. Contrairement aux autres objets, il peut choisir son comportement (c'est ce qui en fait la richesse, mais aussi le rend difficile à manipuler). Un agent peut être un humain, un programme, une interface, ...

- *But* : Un but est un objectif non-opérationnel que le système doit satisfaire. En effet, il ne peut pas être défini en termes d'états du système, et n'est pas contrôlable par un agent. Un but peut être renforcé ou raffiné (réduit). Dans le cas où il est raffiné, il peut être décomposé en sous-buts de deux manières. Soit via des raffinements-ET, soit via des raffinements-OU. Dans le cas de raffinements-ET, si tous les sous-buts sont satisfaits, alors le but est satisfait. Dans le cas de raffinements-OU, la satisfaction d'un seul des sous-buts sera suffisante.
  Ces deux types de décompositions des buts nous donnent une vue structurée du système sous la forme d'un graphe de décomposition ET/OU acyclique.
  Il arrive souvent, et nous le verrons plus loin, que des buts soient en conflit les uns avec les autres. Les buts concernent les objets auxquels ils se réfèrent.
  Les buts sont classifiés par leur modèle de comportement temporel[9] et par le type d'exigence qu'ils expriment.

- *Exigence* [10] : Une exigence est un but opérationnel, c'est-à-dire un but qui peut être formulé en termes d'états contrôlables par un agent. Il s'agit donc de buts assignables à des agents qui en auront la responsabilité (en opposition avec les autres buts de plus haut niveau). Les buts doivent donc si possible être raffinés en contraintes. Il existe des exigences plus faibles, qui peuvent être temporairement violées sans porter préjudice au système. Dans notre travail, nous ne considèrerons pas ce type d'exigences, assumant qu'aucune exigence ne peut être violée.
  Les exigences peuvent être « opérationnalisées » par des actions ou des objets, en respectant les pré-conditions, post-conditions et conditions de déclenchement des actions, ou les invariants des objets suivant le cas dans lequel on se trouve.

- *Hypothèse* : une hypothèse est un fait concernant les agents de l'environnement du système qui est considéré comme vrai. A l'inverse des buts, les hypothèses n'ont pas besoin d'être raffinées ou renforcées. Elles apparaissent ici comme des assertions auxiliaires du système permettant de prouver le raffinement des buts ou leur « opérationnalisation ».

---

[8] Notons la différence entre une pré-condition et une condition de déclenchement: une action ne s'exécute que si sa pré-condition est vérifiée, tandis que cette même action devra être exécutée si la condition de déclenchement est vraie.
[9] Ces « patterns » seront définis en long et en large plus tard. Ils constituent le cœur de notre travail
[10] Ce terme est préféré à celui de contrainte (constraint) qui, utilisé auparavant, pouvait porter à confusion

- *Scénario* : un scénario est une composition d'actions exécutées par les instances des agents qui leur sont associées. Les actions sont exécutées dans des états du système qui respectent leurs pré-conditions, mais leur exécution respecte aussi les invariants du domaine associés aux objets concernés. Les exécutions de ces actions amènent le système dans des états qui doivent satisfaire leurs post-conditions. Les modes de composition incluent la séquence, mais aussi la répétition et le parallélisme des exécutions.

Ce méta-modèle permet de répondre aux questions de « pourquoi » l'application est développée à travers des buts, contraintes et hypothèses. Il répond aussi à la question de savoir qui est concerné par le produit, à travers le concept d'agent associé à une ou des actions. Les deux niveaux suivants (domaine et instance), qui vont instancier ce modèle, permettrons de mettre le doigt exactement sur « qui » et « pourquoi », mais aussi sur « quoi » (opérations, objets, relations) et « quand » (événements, états, scénario).

Le méta-modèle présenté ci-dessous (figure 2) n'est qu'une portion du méta-modèle complet proposé par KAOS. Nous n'avons pris que les éléments qui seront utilisés dans ce mémoire, tout en préservant la nature même de ce méta-modèle. Nous n'avons pas représenté les cardinalités, par souci de lisibilité. Le méta-modèle complet peut être trouvé dans [VLDD91]

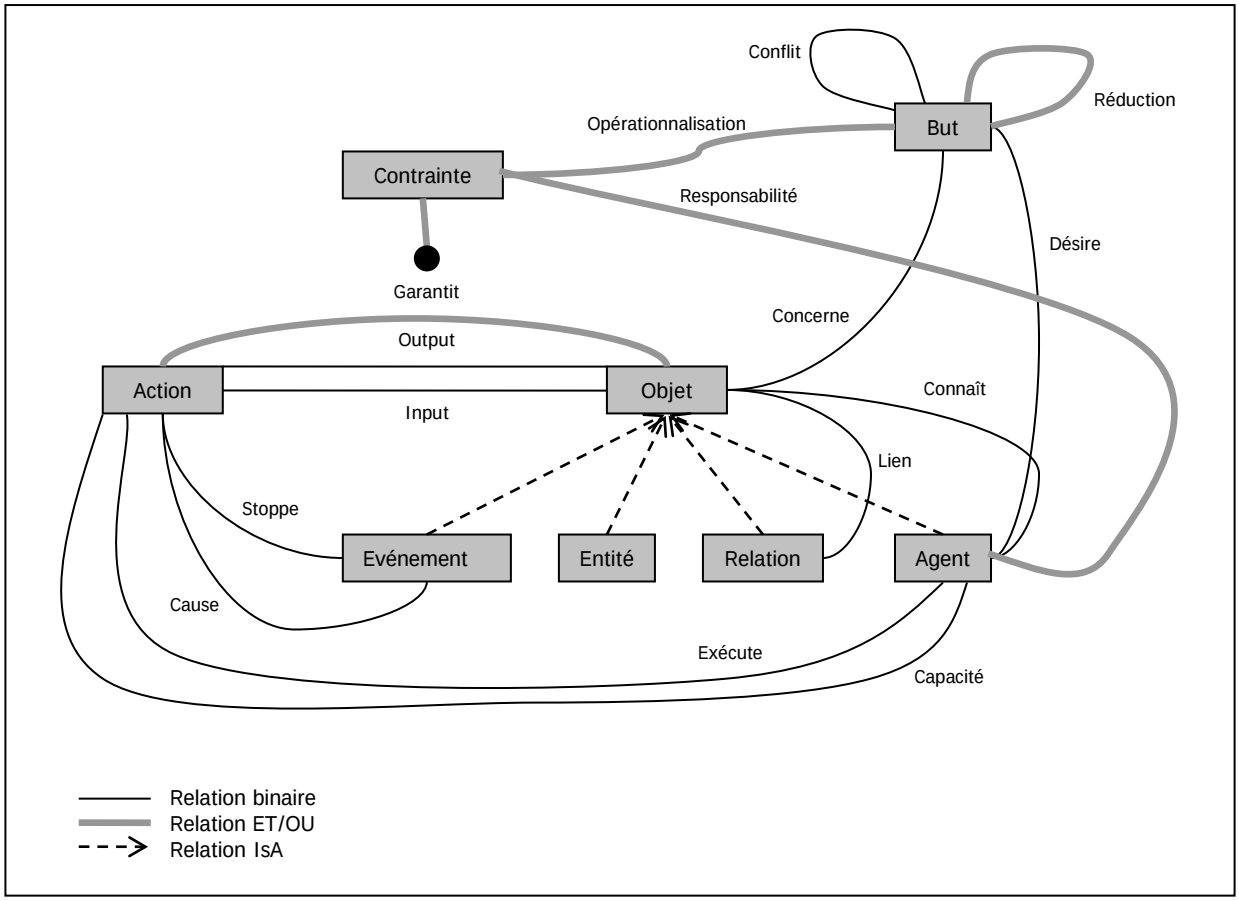


fig. 2.2 - portion du méta-modèle KAOS

2.1.2. Stratégie d'acquisition orientée par les buts (goal oriented)

Fort du méta-modèle, l'acquisition des exigences du système va se faire au travers des instances des méta-concepts relatives au domaine du système.
La stratégie employée pour acquérir les exigences est une approche par les buts que le système doit satisfaire. Elle est constituée des étapes suivantes[11]:

(1) Acquisition de la structure des buts et identification des objets concernés par ces buts

Les buts du système donnés par le « client » (la personne ayant besoin du logiciel, et qui va s'en servir) sont raffinés en une structure de sous-buts, appelée graphe de décomposition (graphe ET/OU). Cette étape est nécessaire car il est très difficile de travailler directement sur les buts du système pour obtenir les exigences. En identifiant ces buts, il sera important de veiller à minimiser les coûts de réduction, ainsi que les conflits entre buts.

(2) Identification préliminaire des agents potentiels du système et des actions associées à ces agents

Il s'agit d'une première identification des agents du système, de leur « catégorie » (humain, machine, programme) et des actions qu'il peuvent effectuer. Ceci permet d'aider le processus de réduction des buts (voir étape précédente) et de guider processus d'opérationnalisation (étape suivante).

(3) Opérationnalisation des buts en contraintes

C'est à cette étape que l'on obtient les exigences du systèmes, ou du moins une bonne partie d'entre elles. Partant du graphe de décomposition des buts et des agents identifiés aux étapes précédentes, les buts sont « opérationnalisés », c'est-à-dire reformulés en termes d'objets et actions réalisables par les agents.

(4) Raffinement des objets et des actions

Après ces trois étapes principales, il se peut, et c'est souvent le cas, que des choses manquent. Ainsi, les exigences formulées à l'étape (3) peuvent inclure de nouveaux objets et actions, ou de nouvelles caractéristiques pour les objets et actions existant déjà. Ceux-ci sont correctement identifiées et définis à cette étape.

(5) Dérivation des conditions portant sur les objets et actions pour prendre en compte dans les exigences

L'étape (4) peut amener l'identification de nouveaux objets et/ou actions non apparentes à l'étape (3). Ceux-ci peuvent intervenir dans les exigences existantes. Il est donc nécessaire de vérifier leur implication et, si besoin est, modifier les exigences pour qu'elles en tiennent compte.

[11] le détail complet peut être obtenu dans [DVLF93].

(6) Identification des responsabilités supplémentaires des agents

Dans cette étape, les agents (énumérés à l'étape 2) capables de se voir assigner une exigence (but opérationnalisé) sont identifiés pour chaque exigence. Cette étape est vitale pour les choix qui vont être effectués à l'étape suivante.

(7) Assignation des actions aux agents, qui vont en devenir responsables

Dernière étape, faisant appel à presque toutes les autres, l'assignation des actions à des agents particuliers va garantir que les exigences qui opérationnalisent les buts du système seront respectées, à travers le comportement attendu des agents.

Cette stratégie comporte des étapes qui peuvent entrer en concurrence. Il est possible de revenir en arrière (backtracking – les étapes 4 et 5 peuvent être effectuées plusieurs fois d'affilée, l'une renvoyant à l'autre), mais les changements effectués devront être reportés aux étapes qui suivent celle qui les comportent. Des tactiques pour la réalisation de ces étapes sont définies par [DVLF93].

Il est très important de bien faire la distinction entre les éléments appartenant à l'environnement interne à l'application développée, et ceux qui sont externes. Ainsi, on évitera d'associer un agent à une action portant sur des objets sortant du domaine interne au système sur lequel on travaille.

Les composantes, une fois clairement identifiées, sont spécifiées comme suit:

- **Goal** NameOfPattern [NameOfGoal]
  Concerns (NameOfObject)*
  RefinedTo (NameOfSubgoal and/or Requirement and/or Assumption)*
  Refines (NameOfGoal)*
  InformalDef
  FormalDef

- **Agent** NameOfAgent
  (**Attribute** NameOfAtttribute : TypeOfAttribute)*
  CapableOf (NameOfAction)*
  Has (NameOfObject)*

- **Relationship** NameOfRelation
  Links (Role x Cardinality)*
  DomInVar

- **Entity** NameOfEntity
  (**Has** NameOfAtttribute : TypeOfAttribute)*

- **Requirement** NameOfRequirement
  Refines NameOfGoal (,NameOfGoal)*
  FormalDef

- **Action** NameOfAction
  Input (NameOfObject)*
  Output (NameOfObject)*
  DomPre
  DomPost

```
        DomTrig
        CapableOf (NameOfAgent)*
```

- ```
  Assumption NameOfAssumption
      Refines NameOfGoal (,NameOfGoal)*
      FormalDef
  ```

Comme on peut le voir, chaque instance des différents concepts est clairement définie. Les objets sont spécifiés de manière formelle, à l'aide d'un langage de spécification utilisant la logique temporelle (cfr. FormalDef). Il en va de même pour les pré-, post-conditions et conditions de déclenchement des actions, ainsi que pour les invariants des relations. Nous reviendrons sur la logique temporelle un peu plus loin, et en détails dans le chapitre 4, qui lui est exclusivement consacré. Ceci apporte un autre niveau de définition des concepts, en plus du niveau déclaratif se rapportant aux noms et attributs.

2.1.2. Instanciation au projet

La dernière étape consiste à travailler sur le niveau des instances. Ce niveau permettra de faire le lien entre les instances du domaine, qui sont encore assez générales, et le système précis dans lequel l'application ou le produit va être développé. Ce « portage » est assez simple à effectuer, les différents concepts ayant déjà été bien instanciés lors de l'étape précédente.

Dans le cadre de notre mémoire, nous ne nous porterons pas sur ce niveau. Notre réflexion porte sur le niveau « domaine », mais les exemples présentés et l'utilisation l'application se font bien au niveau « instance ». Le fait de travailler au niveau « domaine » nous permet de rester dans un cadre général, nous permettant de développer une solution facilement portable, non restreinte à un seul cas de figure, ce qui s'avèrerait inintéressant.

2.3. Introduction à la logique temporelle

Il nous semble ici essentiel de jeter les bases de la logique temporelle, pour permettre au lecteur de pouvoir appréhender dans les meilleures conditions les différentes illustrations des concepts KAOS qui sont présentés par la suite. L'objet de cette section n'est donc pas une analyse sémantique poussée (un chapitre y sera consacré plus loin) mais plutôt une présentation des quelques concepts essentiels, ainsi que leur signification « intuitive ».

Nous verrons donc comment KAOS « utilise » la logique temporelle, via un système de « patterns ». Pour une définition plus exhaustive, nous renvoyons le lecteur au chapitre 4, « Logique Temporelle ».

2.3.1. La logique temporelle en tant que langage de spécification

La définition / manipulation d'objets dans le langage KAOS est structurée au sein d'un modèle de spécification qui lui est propre, utilisé pour définir tant formellement qu'informellement les objets de l'environnement considéré, et les concepts représentés par ces objets.
  ```

A ce niveau, il est essentiel d'obtenir une définition qui soit la plus claire et lisible possible, permettant donc de raisonner sur les objets, buts, hypothèses et agents de manière aisée. Cependant, Cette définition doit être formelle, complète, univoque et non ambiguë. Un langage de spécification est donc nécessaire, tant pour définir l'état du (des) objet(s) / concept(s) manipulé(s) que son (leur) comportement.

Un exemple de cette structure de spécification peut être trouvé dans [Lam&Let]. Il concerne la modélisation d'un système de gestion des urgences médicales, lui-même repris du London Ambulance System[12]. On y trouve cette spécification de but :

Goal Achieve [AmbulanceMobilization]
Concerns Call, Ambulance, Incident
Refines AmbulanceIntervention
RefinedTo IncidentFiled, AmbulanceAllocated, AllocatedAmbulanceMobilized
InformalDef For every responded call about an incident, an ambulance able to arrive at the incident scene within 11 minutes should be mobilized. The ambulance mobilization time should be less than 3 minutes [ORCON standard, 3005]
FormalDef $\forall$ cl:Call, inc:Incident
Responded(cl) $\wedge$ About(cl, inc)
$\Rightarrow \Diamond_{\leq 3m} \exists$ a:Ambulance : Mobilized(a,inc) $\wedge$
$\bullet$[Available(a) $\wedge$ TimeDist(a.Loc,inc.Loc) $\leq$ 11]

Cette spécification définit donc un concept de type But (goal) de haut niveau. Le pattern utilisé est de type Achieve. Nous verrons plus loin les types de patterns différents dont nous disposons.

Cette définition fait également références à d'autres objets / concepts de l'environnement. Citons Call, Ambulance et Incident en tant que concepts de l'environnement, AmbulanceIntervention en tant que but / comportement de plus haut niveau, IncidentFiled, AmbulanceAllocated, AllocatedAmbulanceMobilized en tant que buts plus opérationnels, obtenu par raffinements.

Enfin, On fait également appel à un certain nombre de prédicats définis sur les objets de l'environnement : Responded, About, Mobilized…

La définition formelle, complète, univoque et non ambiguë dont il était question plus haut se retrouve bien ici, sous forme d'une formule de logique temporelle.

[12] [LAS93]. Voir aussi : the London Ambulance System home page, http://hsn.lond-amb.sthames.nhs.uk/http.dir/service/organisation/ featurs/info.html

2.3.2. Les opérateurs

Les opérateurs généralement utilisés dans le langage KAOS sont au nombre de 8[13]. Il s'agit de :

□ : A l'étape suivante. Dépend généralement de la granularité utilisée. Nous verrons plus en détail le concept de granularité lorsque nous évoquerons celui d'histoire.

◇ : Finalement. Garanti que, dans un futur plus ou moins proche, l'événement considéré va se produire.

~ : Toujours. Garanti que l'événement considéré se maintiendra dans le futur.

W : En attendant. Garanti que :

- soit l'événement considéré se maintiendra toujours dans le futur
- soit un événement, défini a priori, se produira dans le futur et mettra un terme au maintient de l'événement premier.

U : Jusque : Garanti que l'événement considéré se maintiendra jusqu'à l'arrivée d'un événement bien précis, défini a priori.

● : A l'étape précédente. Version « passée » du premier opérateur

■: Finalement dans le passé. Garanti que l'événement considéré s'est déjà produit.

■ : Toujours dans le passé. Garanti que l'événement considéré s'est maintenu jusque maintenant.

2.3.3. Sémantique des opérateurs, concept d'histoire

Les assertions formelles sont interprétées à partir de séquences d'états ou histoires. On dira généralement que chaque assertion est validée par certaines séquences, et falsifiée par d'autres. On notera :

```
(H,i) |= P
```

pour exprimer que l'assertion P est satisfaite par l'histoire H, à la position i. Etant donné que nous sommes ici au sein de la logique temporelle, on peut ajouter que $i \in T$, où T dénote une structure linéaire temporelle discrète.

[13] Ils sont basés sur ceux de [MaPn92]

La granularité de cette structure temporelle est fortement dépendante de l'environnement modélisé. C'est elle qui interviendra dans la sémantique des opérateurs □ et ●, désignant respectivement l'état suivant et l'état précédent.

A ce niveau, nous pouvons déjà définir une sémantique des différents opérateurs. Pour une version plus complète de cette sémantique, voir le chapitre 4, relatif à la Logique Temporelle.

Cette sémantique est inspirée des travaux de Pnueli & Manna[14].

Pour les opérateurs futurs :

(H,i) |= □ P	ssi	(H,next(i)) |= P
(H,i) |= ◇ P	ssi	(H,j) |= P pour un j ≥ i
(H,i) |= ~ P	ssi	(H,j) |= P pour tout j ≥ i
(H,i) |= P U Q	ssi	∃ j ≥ i tq (H,j) |= Q
			∧ ∀ k, i ≤ k < j : (H,k) |= P
(H,i) |= P W Q	ssi	(H,i) |= P U Q ∨ (H,i) |= ~ P

Et pour ceux du passé :

(H,i) |= ● P	ssi	(H,previous(i)) |= P
(H,i) |= ■ P	ssi	(H,j) |= P pour un j ≤ i
(H,i) |= □ P	ssi	(H,j) |= P pour tout j ≤ i

On remarquer a d'emblée que certaines assertions ne sont pas calculables / décidables en pratique. En effet, si l'on admet que T est une structure non bornée (ligne de temps), une assertion de type (H,i) |= □ P est potentiellement invérifiable.

Plus clairement : ∀ x ≥ i tq (H,x) |= □ P, ∃ toujours (x+1) qui peut éventuellement satisfaire à (H,(x+1)) |= P, sans que cela soit vérifiable.

Intuitivement, on dira que l'on ne peut avoir aucune garantie sur le déroulement d'une histoire si on accepte que la structure T soit dépourvue de borne temporelle. C'est pourquoi, en pratique, on utilisera les opérateurs temps-réel suivants :

(H,i) |= $\Box_{\leq d}$ P	ssi	(H,j) |= P pour un d ≥ j ≥ i
(H,i) |= $\sim_{\leq d}$ P	ssi	(H,j) |= P pour tout d ≥ j ≥ i

[14] [MaPn92]

Cette borne temporelle pouvant être bien entendu appliquée à tous les opérateurs, à l'exception des □ et ●, où elle perd tout son sens.

2.3.4. Les patterns, usage pratique de la logique temporelle dans KAOS

En pratique, on utilisera dans KAOS des formules-types ou patterns pour définir le comportement des objets de l'environnement considéré. Ces patterns sont :

`Achieve : C ⇒ □ T`
La condition de départ C implique la réalisation de l'objectif T dans un futur plus ou moins proche.

`Cease : C ⇒ □ ¬ T`
La condition de départ C implique la cessation, au terme d'un futur plus ou moins proche, de l'assertion T

`Maintain : C ⇒ ~ T`
La condition de départ C implique le maintient de l'assertion T à partir de l'instant courant. C'est une expression d'un invariant.

`Avoid : C ⇒ ~ ¬ T`
La condition de départ C implique l'interdiction de l'assertion T à partir de l'instant courant.

2.4. Analyse d'obstacles avec KAOS

Deux attitudes sont susceptibles d'être suivies lors de la vérification de programmes. Ces attitudes sont a priori (et heureusement) équivalentes. Nous avons donc le choix entre *vérifier* que tout fonctionne comme prévu et *détecter* les cas de fonctionnement anormal.

La vérification consiste à définir des buts, exprimés par exemple en logique temporelle, et présentant des conditions sur les évènements du programme. Ces buts sont des situations à atteindre, à maintenir, à éviter… Dans ce cas, nous devons nous assurer à chaque instant de la concordance entre l'état du système et les buts spécifiés.

La détection consiste à dériver une série d'obstacles aux buts spécifiés. Il faut ensuite détecter ces situations dans lesquelles les spécifications du programme sont prises en défaut. Ce chapitre présente donc un ensemble de concepts théoriques liés aux obstacles, des méthodes permettant de dériver des obstacles à partir des buts, et enfin des méthodes de résolution de ces obstacles.

Notons cependant que la détection d'obstacles n'est pas la méthode choisie dans le cadre de l'outil Orka. Ce dernier utilise plutôt l'autre méthode, à savoir la vérification des buts.

2.4.1. Introduction

D'un point de vue sémantique, on peut considérer un but comme définissant un ensemble de comportements désirés, et un comportement désiré comme étant une

séquence temporelle d'états du système ou histoire. Une histoire positive est une séquence de transitions d'états qui produit un comportement désiré.

Par comparaison, on dira qu'un obstacle dénote un ensemble de comportements non désirés. Une histoire sera qualifiée de négative si elle produit un comportement inclus dans cet ensemble.

a. Définition :

Plus formellement, définissons *G* un but, et *Dom* un ensemble de propriétés du domaine. Une assertion *O* est qualifiée d'obstacle si et seulement si : [Lam&Let]

```
1. {O, Dom} |= ¬ G            (obstruction)
2. {O, Dom} |= false          (domain-consistency)
```

Ce qui signifie que :

1. La négation du but est la conséquence logique, étant donné les propriétés du domaine, de l'assertion *O*.

2. L'obstacle n'est pas en contradiction avec les propriétés du domaine. Il existe donc une histoire `H` telle que `H |= O`

Illustrons ces propos par un exemple. Considérons une bibliothèque et un but de haut niveau, selon lequel toute requête d'emprunt devrait être satisfaite tôt ou tard :

```
Goal Achieve[BookRequestSatisfied]
     FormalDef : ∀ bor:Borrower, b:Book
                 Requesting(bor,b)
                 ⇒ □(∃ bc:BookCopy)[Copy(bc,b) ∧ Gets(bor,bc)]
```

Un obstacle à ce but peut être formalisé comme suit :

```
∃ bor:Borrower, b:Book
□{Requesting(bor,b) ∧
~(∀ bc:BookCopy) [Copy(bc,b) ⇒ ¬ Gets(bor,bc)]}
```

Ce qui correspond bien à une assertion où la requête d'emprunt ne sera pas honorée. Le scénario amenant à cette assertion est celui de la famine, où, chaque fois qu'une copie devient disponible, elle est allouée à un autre agent.

b. Complétude :

Cet obstacle est un exemple parmi d'autre. En règle générale, à un but correspond plus d'un obstacle. Une propriété essentielle est alors à considérer. Il s'agit de la complétude (domain-completeness).

Un ensemble d'obstacles à un but G est réputé complet par rapport au domaine (domain-complete) si et seulement si :

$$\{\neg O_1, \ldots, \neg O_n, Dom\} \models G$$

Ce qui signifie que si aucun des obstacles n'est rencontré, alors on peut certifier que le but est atteint. Cette notion est essentielle au sens où elle garantit l'équivalence de deux approches, à savoir la surveillance des buts et la détection des obstacles. La surveillance des buts, orientation que nous avons prise dans le cadre de ce mémoire, garantit que tout manquement aux buts spécifiés sera détecté par le système, quel que soit l'obstacle rencontré. A contrario, la détection d'un ensemble d'obstacles relatifs à un but ne garantira la détection de tout manquement aux buts spécifiés que si, et seulement si, l'ensemble considéré possède ladite propriété de complétude.

c. Raffinement :

La nature des obstacles n'étant pas différente de celle des buts, ceux-ci peuvent (doivent) être également raffinés. Tout comme pour les buts, on distingue les « raffinements-ET » et les « raffinements-OU ».

Un ensemble d'obstacles est un raffinement-ET de l'obstacle *O* si et seulement si :

1. $\{O_1, \ldots, O_n, Dom\} \models O$ (entailment)
2. $\{O_1, \ldots, O_n, Dom\} \models false$ (consistency)

Deux propriétés qui peuvent être complétées par une troisième :

3. $\forall\ i : \{\wedge_j\ {}_iO_j, Dom\} \models O$ (minimality)

Pour le raffinement-OU, nous aurons :

1. $\forall\ i : \{O_i, Dom\} \models O$ (entailment)
2. $\forall\ i : \{O_i, Dom\} \models false$ (consistency)
3. $\{\neg\ O_1, \ldots, \neg\ O_n, Dom\} \models \neg\ O$ (completeness)

Auxquelles nous pourrons rajouter également :

4. $\forall$ i, j : $\{O_i, O_j, Dom\} \models false$ (disjointness)

d. Classification d'obstacles :

Comme il existe des classes différentes de but, il existe les classes correspondantes d'obstacles. Nous pouvons donc définir :

- Les obstacles de non-satisfaction (Non-Satisfaction Obstacles), qui empêchent la satisfaction des requêtes des agents (Satisfaction Goals)
- Les obstacles de non-information (Non-Information Obstacles), qui empêchent les agents d'être informés sur les états des objets de leur environnement (Information Goals)
- Les obstacles d'imprécision (Inaccuracy Obstacles) qui empêchent la cohérence entre les états des objets dans l'environnement et leur représentation au sein du système (Accuracy Goals)
- Les dangers (Hazard Obstacles) qui empêche le bon déroulement des Safety Goals
- Les menaces (Threat Obstacles) qui empêchent le bon déroulement des buts de sécurité (Security Goals)

Ces classes peuvent bien entendu être précisées par la suite. La connaissance de ces classes permet de disposer de bases de travail pour une identification heuristique des buts et des obstacles qui en sont dérivés.

2.4.2. Génération des obstacles

Comme nous l'avons vu plus haut, la génération d'obstacle est un procédé en deux phases :

1. Etant donné un but défini par une assertion, trouver une assertion qui peut empêcher ce but.

2. Vérifier que cette dernière assertion (candidat obstacle) est cohérente avec le domaine d'application.

Concernant le deuxième point, on peut utiliser, au choix, des techniques de vérification déductives [15] (consistency), ou élaborer des scénarii ou histoires satisfaisant ces assertions (feasibility).

Ce genre de vérification, en ce compris toute la théorie du model checking, ne sera pas abordé ici.

Concernant le premier point, nous avons identifié trois méthodes permettant la dérivation systématique des obstacles, au départ des buts. Rappelons que, s'il est souhaitable de disposer de méthodes systématiques de dérivation d'obstacles, il est également souhaitable que celles-ci soient les plus systématiques possibles, et qu'elles garantissent certaines propriétés essentielles (notamment la complétude).

- L'usage du calcul formel pour la négation des buts.
- L'usage de patterns d'obstructions.

[15] [MaSte96], [ORS95]

- L’usage d’identifications heuristiques basées sur la classification des obstacles.

a. La négation des buts :

Etant donné un but G, ce principe consiste en un calcul des préconditions pour obtenir la négation ¬ G à partir du domaine d’application. Chaque précondition obtenue étant un candidat-obstacle. Illustrons par un exemple.

Soit un système de type Meeting Scheduler, et un but de haut niveau exprimant le fait que des personnes au courant d’une réunion, qui les intéresse et pour laquelle leurs contraintes d’horaire sont compatibles, devraient participer à cette réunion.

La formalisation de ce but est la suivante :

Goal Achieve[InformedParticipantsAttendance]
FormalDef : ∀ m:Meeting, p:Participant
Intended(p,m) ∧ Informed(p,m) ∧
Convenient(p,m)
⇒ □ Participates(p,m)

L’étape d’initialisation de la régression consiste à prendre la négation du but défini ci-dessus. Nous avons donc :

∃ m:Meeting, p:Participant
Intended(p,m) ∧ Informed(p,m) ∧ Convenient(p,m) ∧
~ ¬ Participates(p,m) (1)

Qui représente notre candidat-obstacle.

Supposons maintenant que le domaine d’application contient la propriété suivante :

∀ m:Meeting, p:Participant :
Participates(p,m) ⇒ Holds(m) ∧ Convenient(p,m)

Propriété pouvant être également exprimée via sa contraposée : (2)

∀ m:Meeting, p:Participant :
¬ [Holds(m) ∧ Convenient(p,m)] ⇒ ¬ Participates(p,m)

Si l'on substitue (¬ Participates(p,m)) dans la formule (1) par sa représentation dans (2), nous obtenons :

$$\exists\ m{:}Meeting,\ p{:}Participant$$
$$Intended(p,m) \wedge Informed(p,m) \wedge Convenient(p,m)$$
$$\wedge \sim [\neg\ Holds(m) \vee \neg\ Convenient(p,m)]$$

Qui est notre nouveau candidat-obstacle. Il recouvre deux situations : celle d'une réunion qui n'est jamais organisée, et celle d'une date qui ne convient plus à un participant. On pourra dès lors raffiner cet obstacle en deux sous-obstacles (raffinement-OU) :

- MeetingPostponedIndefinitely
- LastMinuteImpediment

De façon plus générale, nous pouvons définir la procédure de régression comme suit :

1. Etant donné un but *G*, définir *O* comme $\neg\ G$
2. Soit $A \Rightarrow C$, une règle du domaine d'application, avec un matching positif entre *C* et une sous-formule *L* de *O*, on défini :

$$\mu = mgu\ (L,C)$$
$$O = O[L/A.\mu]$$

Chaque itération produit alors un candidat-obstacle de plus en plus précis / spécifique. C'est à l'utilisateur de décider quand arrêter le processus, sachant que l'obstacle généré doit être assez pertinent et précis pour d'une part identifier les scenarii qui le satisfont, et, d'autre part, identifier les différentes méthodes et stratégies de résolutions d'obstacles qui conviendraient au cas présent.

Cette procédure repose sur les définitions et notations suivantes :

- Pour une formule $\varphi(u)$, contenant une ou plusieurs occurrences de u, une occurrence de u est réputée positive dans φ si elle n'apparaît pas dans une sous-formule de forme $p \leftrightarrow q$ et est incluse dans un nombre pair de négations.
- $mgu(F_1,F_2)$ dénote l'unificateur le plus général de F_1 et F_2, ce qui correspond dans le cas présent à une substitution des littéraux des formules permettant de les rendre syntaxiquement équivalentes.
- F. μ représente l'application d'une substitution sur la formule F
- F $[L_1/L_2]$ représente le remplacement de chaque occurrence de L_1 par L_2 dans la formule F

b. L'utilisation de patterns d'obstruction :

Comme nous l'avons vu plus haut, le langage KAOS utilise la logique temporelle via des patterns définissant le comportement des différents objets de l'environnement. Il peut dès lors paraître vain et inutile d'avoir recours à une

procédure complexe de régression pour la génération d'obstacle, alors que l'on peut définir, pour chaque pattern générique, un ensemble d'obstacles génériques qu'il suffit d'instancier suivant les données à considérer dans le cas d'utilisation courant. Ceci peut donc être vu comme une procédure simplifiée, où l'utilisateur a à sa disposition un ensemble de patterns de buts « courants » et l'ensemble d'obstacles correspondants à ces buts. Son travail se limite alors à un simple remplacement des littéraux (C, T, P, M, B dans l'exemple) par les évènements concrètement observés dans son système.

Dans ce cas, la correction des patterns d'obstruction est prouvée une fois pour toute, et est indépendante de l'instanciation.

Chaque pattern est ici représenté par un arbre de raffinement où la racine représente la négation du but considéré, et où les feuilles sont des assertions / propriétés du domaine d'application.

Illustrons ces principes par des exemples :

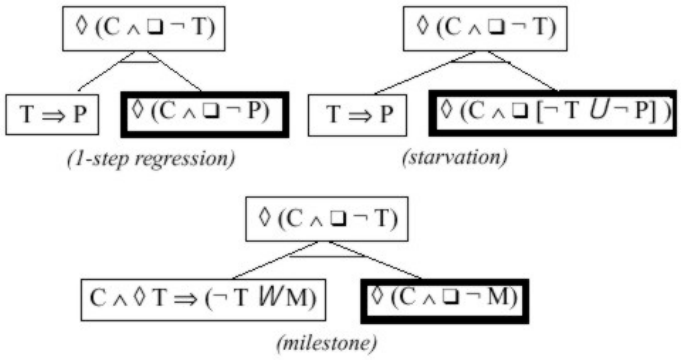


Fig.4 - AND-refinement patterns for obstacles to the goal C ⇒ ◊ T

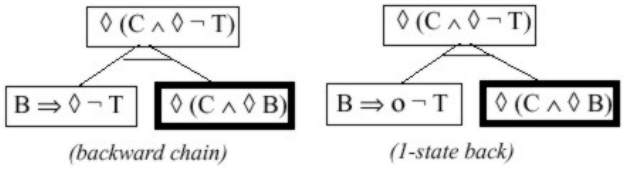


Fig.5 - AND-refinement patterns for obstacles to the goal C ⇒ ❑ T

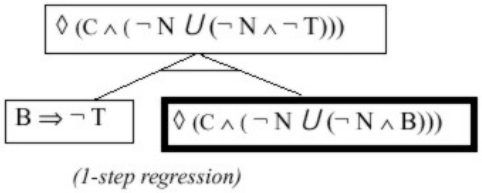


Fig. 6 - AND-refinement pattern for obstacles to the goal C ⇒ T W N

fig. 2.3 –arbres de raffinement ET

Chaque racine représente donc la négation du but considéré. Il s'agit en l'occurrence ici des patterns Achieve, Maintain et Unless. Le fils gauche représente la propriété considérée du domaine d'application. Le fils droit

correspond au pattern générique correspondant. On peut donc voir que pour un pattern Achieve, nous considérons ici trois obstacles, correspondant à :

- Une régression en une étape, correspondant à l'intégration d'une propriété du domaine.
- Une famine.
- Un Millestone.

De façon plus exhaustive, nous avons, pour un but $(R \Rightarrow \Diamond S)$ de type Achieve :

	assertion	subobstacle
1-step regress	$S \Rightarrow P$	$\Diamond [R \wedge \Box \neg P]$
	$S \Rightarrow P$	$\Diamond [R \wedge (\neg S \mathcal{U} \Box \neg P)]$
starvation	$S \Rightarrow P$	$\Diamond [R \wedge \Box (\neg S \mathcal{U} \neg P)]$
missing source	$R \wedge \Diamond S \Rightarrow P$	$\Diamond [R \wedge \neg P]$
non-persistence	$R \wedge \Diamond S \Rightarrow P \mathcal{W} S$	$\Diamond [R \wedge \neg S \mathcal{U} (\neg P \wedge \neg S)]$
non-persistence	$R \wedge \Diamond S \Rightarrow P \mathcal{W} (P \wedge S)$	$\Diamond [R \wedge (\neg S \mathcal{U} \neg P)]$
milestone	$R \wedge \Diamond S \Rightarrow \neg S \mathcal{W} M$	$\Diamond [R \wedge \Box \neg M]$
blocking	$B \Rightarrow \Box \neg S$	$\Diamond [R \wedge (\neg S \mathcal{U} B)]$
substitution	$S' \Rightarrow \Box \neg S \wedge \blacksquare \neg S$	$\Diamond [R \wedge \Diamond S']$
strengthening	$R \wedge \Diamond S \Rightarrow \Diamond [P \wedge (P \mathcal{W} S)]$	$\Diamond [R \wedge \Box \neg P]$
starvation	$R \wedge \Diamond S \Rightarrow \Diamond [P \wedge (P \mathcal{W} S)]$	$\Diamond [R \wedge (\neg S \mathcal{U} \Box \neg P])]$
	$R \wedge \Diamond S \Rightarrow \Diamond [P \wedge (P \mathcal{W} S)]$	$\Diamond [R \wedge (\neg S \mathcal{U} (\neg S \wedge \Box \neg P))]$

fig 2.4 - tableau des patterns d'obstruction

Pour ce même type de pattern Achieve, nous pouvons également définir des arbres de raffinement-OU de patterns d'obstruction. Pour le Millestone, par exemple, nous avons un arbre-ET, où le fils gauche représente toujours une propriété du domaine d'application, et où le fils droit est raffiné en deux sous-obstacles alternatifs :

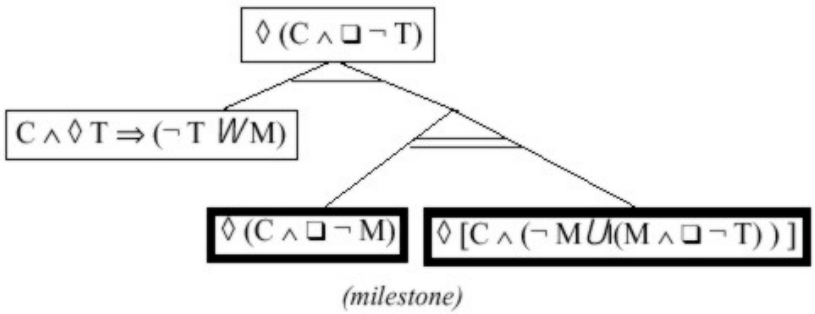


fig. 2.5 – arbre de raffinement OU

Et, exhaustivement, toujours pour $R \Rightarrow \Diamond S$:

assertion	obstacle	obstacle	obstacle
$S \Leftrightarrow P \wedge Q$	$\Diamond [R \wedge \Box \neg P]$	$\Diamond [R \wedge \Box \neg Q]$	$\Diamond [R \wedge \Diamond P \wedge \Diamond Q \wedge \Box \neg (P \wedge Q)]$
$S \Rightarrow P$	$\Diamond [R \wedge \Box \neg P]$	$\Diamond [R \wedge \Diamond P \wedge \Box \neg S]$	
$S \Rightarrow P$	$\Diamond [R \wedge \neg S U \Box \neg P]$	$\Diamond [R \wedge \Diamond P \wedge \Box \neg S]$	
$S \Rightarrow P$	$\Diamond [R \wedge \Box (\neg S U \neg P)]$	$\Diamond [R \wedge \Diamond (P W (P \wedge S)) \wedge \Box \neg S]$	
$R \wedge \Diamond S \Rightarrow P$	$\Diamond [R \wedge \neg P]$	$\Diamond [R \wedge P \wedge \Box \neg S]$	
$R \wedge \Diamond S \Rightarrow P W S$	$\Diamond [R \wedge \neg S U (\neg P \wedge \neg S)]$	$\Diamond [R \wedge P W S \wedge \Box \neg S]$	
$R \wedge \Diamond S \Rightarrow P W (P \wedge S)$	$\Diamond [R \wedge \neg S U \neg P]$	$\Diamond [R \wedge \Box \neg S \wedge P W (P \wedge S)]$	
$R \wedge \Diamond S \Rightarrow \neg S W M$	$\Diamond [R \wedge \Box \neg M]$	$\Diamond [R \wedge \neg M U (M \wedge \Box \neg S)]$	
$B \Rightarrow \Box \neg S$	$\Diamond [R \wedge \neg S U B]$	$\Diamond [R \wedge \Box \neg S \wedge \neg B W S]$	
$P \Rightarrow \Box \neg S \wedge \blacksquare \neg S$	$\Diamond [R \wedge \Diamond P]$	$\Diamond [R \wedge \Box \neg S \wedge \Box \neg P]$	

fig. 2.6 - tableau des patterns d'obstruction

c. L'utilisation d'heuristiques d'identification :

Comme nous l'avons évoqué au point 2.1.4., il existe des classes d'obstacles. Partant de ce constat, il est parfois intéressant de substituer des méthodes heuristiques aux techniques formelles. Ces heuristiques sont souvent plus simples à utiliser, plus souples et plus rapides, mais les résultats obtenus sont parfois moins précis, et il n'y a pas de garantie formelle de correction ou de complétude.

La structure générale de ces heuristiques est de la forme : « Si la spécification a telle ou telle caractéristique, Alors il faut considérer tel ou tel type d'obstacle. ». On distingue des heuristiques générales, non liées spécifiquement à une classe bien définie d'obstacle, et des heuristiques spécifiques, fortement dépendantes des classes susnommées.

Les heuristiques générales réfèrent uniquement au méta-modèle KAOS. En voici quelques illustrations :

> **SI** un ***Agent*** doit *Monitorer / Contrôler* un ***Objet*** pour garantir un ***But*** auquel il est *Assigné*, **ALORS** on doit considérer les ***Types d'Obstacles*** suivants :
>
> - InfoUnavailable : L'information nécessaire concernant l'état de ***l'Objet*** fait défaut.
> - InfoNotInTime : L'information nécessaire concernant l'état de ***l'Objet*** est disponible trop tard.
> - WrongBelief : L'information nécessaire concernant l'état de ***l'Objet*** enregistrée dans la mémoire de ***l'Agent*** diffère de l'état actuel de ***l'Objet***.
>   Cet obstacle pouvant par ailleurs être raffiné à nouveau en :
>   - WrongInfoProvided
>   - InfoCorrupted
>   - InfoOutDated
>   - InfoForgotten
>   - WrongInference
>   - InfoConfusion
>   Dernier point pouvant à son tour être également raffiné :
>   - InstanceConfusion
>   - ValueConfusion
>   - UnitConfusion

Les heuristiques spécifiques, quant à elles, ont une forme dépendante de la classe d'obstacles à laquelle elles se réfèrent. Par exemple :

> **SI** on considère un ***But*** *MessageDelivered* de la classe **InformationGoal**, **ALORS** on doit considérer des ***Obstacles*** tels que *MessageUndelivered*, *MessageDeliveredAtWrongPlace*, *MessageDeliveredAtWrongTime*, *MessageCorrupted*,…
>
> **SI** on considère un ***But*** de la classe **StimulusResponse**, Alors on doit considérer les ***Types d'Obstacles*** suivants :
>
> - *StimulusIgnored*, *TooLatePickUp*, *IncorrectValue* ou *StimuliConfused* relatifs au but abstrait ***StimulusPickedUp***.
> - *NoResponse*, *ResponseTooLate*, *ResponseIgnored* ou *WrongResponse* relatifs au but abstrait ***ResponseProvided***.

2.4.3. Résolution d'obstacles

Le processus de résolution d'obstacle couvre en réalité deux aspects. L'un, technique, est celui de la génération de solutions candidates à partir d'une situation d'obstacle identifiée. L'autre, couvrant des aspects aussi bien techniques qu'économiques, est la sélection d'une solution, d'un comportement, choisi au sein des solutions générées. C'est le premier point qui nous intéresse ici. Nous

allons donc passer en revue des stratégies permettant de générer des solutions au problème (obstacle) rencontré.

Trois classes de stratégies sont à notre disposition. Il s'agit de savoir si l'obstacle doit être *éliminé*, *réduit* ou *toléré*. Il résultera de l'application de ces stratégies une structure de buts transformée, mais également des spécifications transformées et, parfois, un domaine d'application transformé.

Entendons-nous bien : nous sommes ici encore au sein du modèle Kaos, et nous définissons ici des modifications à y apporter. Il ne s'agit donc pas de stratégies à appliquer « at runtime », pour modifier le comportement d'un programme déjà spécifié et écrit. Nous sommes donc encore à la description du comportement du programme.

a. L'élimination d'obstacle :

Pour éliminer un obstacle, deux conditions possibles : l'obstruction doit être empêchée ou l'obstacle doit être rendu incohérent / infaisable par rapport au domaine d'application.

Les sous-stratégies à notre disposition sont :

La substitution de buts :

Il s'agit ici d'identifier une branche alternative de raffinement de buts, pour un but de plus haut niveau, branche dans laquelle le but et l'obstacle incriminés ne seraient plus présents. Dans le cas d'un transfert d'information par e-mail, on peut raffiner le but de plus haut niveau InfoKnown par la branche connue (information propagée par e-mail) et également par une autre (information propagée par courrier classique). Cette redondance permet d'éliminer l'obstacle EmailUnavailable.

La substitution d'agents :

On considère ici l'assignation alternative d'agents pour prévenir un scénario d'obstacle. L'obstacle EmailNotChecked peut être invalidé si l'agent assigné n'est plus le patron mais son secrétariat.

La prévention d'obstacle :

Soit un but *G* de forme générale ~GC (GoalCondition), l'obstacle relatif *O* a la forme générale □OC (ObstacleCondition). Par exemple, l'obstacle à (~ x) sera (□¬ x) et l'on fait la distinction entre l'obstacle (□ ¬ x) et la condition d'obstacle (¬ x) . Pour prévenir l'apparition de cet obstacle, on peut ajouter à la structure de buts le but suivant : G* : ~ ¬ OC. Il se peut également que ce but G* soit en fait, en toute généralité, une propriété manquante du domaine d'application. Dans ce cas, il doit y être réintégré.

L’anticipation d’obstacles :

Soit notre obstacle O. S’il s’avère que OC s’exprime sous la forme :

$$OC \Rightarrow \Box_{\leq d} P$$

On peut introduire un sous-but G* qui empêchera l’apparition de O :

$$G^* : P \Rightarrow \Box_{\leq d} \neg P$$

La dé-idéalisation des buts :

L’obstruction d’un but par un obstacle peut provenir du fait que ce but est trop idéalisé. Le principe est ici de transformer le but en un but nouveau, équivalent en pratique, et invalidant l’obstacle. On peut à cet effet utiliser des patterns de dé-idéalisation. En voici quelques exemples :

goal	obstacle	deidealized goal
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge \neg P]$	$R \wedge P \Rightarrow \Diamond S$
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge \Box \neg P]$	$R \wedge (P \mathcal{W} S) \Rightarrow \Diamond S$
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge (\neg S \mathcal{U} \neg P)]$	$R \wedge (P \mathcal{W} S) \Rightarrow \Diamond S$
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge (\neg S \mathcal{U} \Box \neg P)]$	$R \wedge \Box \Diamond P \Rightarrow \Diamond S$
$R \Rightarrow \Diamond S$	$\Diamond [R \wedge \Box (\neg S \mathcal{U} \neg P)]$	$R \wedge \Diamond (P \mathcal{W} (P \wedge S)) \Rightarrow \Diamond S$

fig. 2.7 – tableau de dé-idéalisation

La transformation de domaine :

Cette stratégie consiste en une transformation du domaine d’application. L’ensemble des propriétés du domaine est modifiée pour rendre incohérent l’obstacle, et donc l’éliminer.

b. La réduction d’obstacle :

Il s’agit ici de réduire le nombre d’occurrences des obstacles et non de les éliminer. Les cas les plus courants sont ceux relatifs à la motivation des agents humains du système. On utilise des mécanismes de dissuasion ou de récompense, sans pour autant annihiler la capacité de l’agent à générer l’obstacle.

c. La tolérance aux obstacles :

Il est ici question de stratégies à mettre en œuvre pour contourner des obstacles ne pouvant pas être totalement empêchés, que ce soit pour des raisons techniques ou économiques. Ces stratégies sont :

La restauration des buts :

Soit G un but, et O, un obstacle à ce but. On défini G*, garantissant que si O est rencontré, alors G sera satisfait à nouveau plus tard.

$$G^* : OC \Rightarrow \Box G$$

C'est typiquement le cas du renvoi d'un courrier si non réponse.

L'atténuation d'obstacles :

Le principe est ici aussi d'ajouter un nouveau but pour, cette fois ci, atténuer les effets d'un obstacle rencontré. Deux formes d'atténuation sont proposées :

- L'atténuation faible :
  On défini G', version affaiblie du but G considéré. On défini également G*, nouveau but, tel que :

  $$G^* : OC \Rightarrow G'$$

  Le nouveau but garanti donc que, si l'obstacle se produit, on satisfera à une version plus faible, dé-idéalisée, du but G.

- L'atténuation forte :
  Soit G', but de plus haut niveau, parent de G, on défini G*, nouveau but, tel que :

  $$G^* : OC \Rightarrow G'$$

  Le nouveau but garanti donc que, si l'obstacle se produit, on satisfera quand même au but de plus haut niveau G'.

L'absence de réaction :

Il se peut également que certains buts soient si accessoires qu'une gestion des obstacles relatifs à ces buts soit superflue.

2.5. Illustration

Le cas présenté ci-dessous est un exemple classique, il s'agit d'une mine. Cette étude de cas a déjà été abordée par [BuLi91], [MaHa92], [Math96] et [DYP98]. l a également été repris par Christophe Ponsard [Pon98], dans le cadre de sa thèse[16]. Nous allons ici présenter la méthode proposée par KAOS (en reprenant ce qui a été fait par Christophe Ponsard), puis nous reprendrons plus loin cet exemple afin d'illustrer le produit de notre recherche.

2.5.1 Enoncé

Une description complète du problème de la mine peut être trouvé dans [Math96]. Nous avons repris l'adaptation de Christophe Ponsard pour cette illustration.

Considérons un système de contrôle de sécurité d'une mine. La mine est surveillée par trois détecteurs de types différents: un détecteur de niveau d'eau (water level detector - WD), un détecteur de fonctionnement/de disfonctionnement de la pompe à eau (pump failure detector - PD) et un détecteur du niveau de gaz dans la mine (gas detector - GD). WD mesure le niveau d'eau collecté dans un réservoir à l'intérieur de la mine. PD détecte le disfonctionnement de la pompe évacuant l'eau du réservoir hors de la mine. Enfin, GD mesure le niveau de gaz de la mine (du méthane dans ce cas précis). Notons que le réservoir d'eau est situé au plus bas niveau de la mine et collecte l'eau qui tombe. La pompe est utilisée pour vider le réservoir lorsqu'il dépasse un niveau maximum.

Quand le niveau d'eau mesuré par WD dépasse le niveau critique WMAX, le système de contrôle doit mette en marche la pompe à eau, et quand ce même niveau d'eau (mesuré par WD) atteint ensuite le niveau minimum WMIN, le système de contrôle arrête la pompe. Quand le niveau critique MMAX est atteint, et que PD détecte un disfonctionnement de la pompe, une alarme « danger d'inondation » est mise en marche par le système de contrôle, ce qui va provoquer l'évacuation de la mine. Quand le niveau de gaz mesuré par GD atteint le niveau critique GMAX, le système de contrôle lance une alarme « danger de suffocation », qui aura également pour effet de provoquer l'évacuation de la mine. De plus, pour éviter tout risque d'explosion, la pompe d'évacuation d'eau ne doit pas fonctionner lorsque le niveau de gaz de la mine dépasse GMAX.

2.5.2 La méthode d'analyse KAOS

Notons tout d'abord que cet exemple est un peu particulier. En effet, nous analysons un système déjà existant, dont nous avons la spécification (informelle). Cette spécification résulte des choix d'implémentation effectués par les concepteurs du système de contrôle. De plus l'énoncé fait des hypothèses sur le système analysé. KAOS va nous permettre notamment d'identifier ces hypothèses.

---

[16] "Detection of runtime inconsistencies in composite agent systems and their reconciliation with the system specifications", voir http://www.info.ucl.ac.be/people/chp/chp.html

Pour rappel, la méthode utilisée fait appel à plusieurs étapes qui peuvent entrer en concurrence, et il est possible de revenir en arrière. De plus, de par le fait que nous travaillons ici sur un système déjà existant, nous devons faire face à deux problèmes:

- incomplétude: présence d'hypothèses cachées, buts non opérationnalisés, manque d'assignation des responsabilités à des agents pour certaines contraintes,…
- inconsistance: présence de conflits non résolus entre buts, agent dans l'incapacité de faire face à leur responsabilités,…

La méthode proposée par KAOS va nous permettre de lever le voile sur ces problèmes et de les intégrer dans la spécification.

(1) Acquisition de la structure des buts et identification des objets concernés par ces buts

a. Buts de haut niveau

A la lecture de l'énoncé, on s'aperçoit directement qu'un but principal du système[17] est de maintenir la mine sécurisée, c'est-à-dire de maintenir les mineurs en vie. D'autres buts existent mais, à la lecture de l'énoncé et dans le cadre de l'illustration, nous nous limiterons à celui-ci

```
Agent Miner
   Attribute : alive : Boolean

Entity Mine

Relationship Inside between Miner and Mine

Goal Maintain[SecureMine]
   InformalDef : the mine has to be secure
   FormalDef : (∀ p:Miner, m:Mine) Inside(p,m) ⇒  p.alive
```

Le but principal peut être raffiné grâce à l'analyse des obstacles (cfr 2.4). En effet, l'énoncé nous donne 3 propriétés du domaine :

- quand une inondation a lieu dans la mine, les mineurs risquent de mourir de noyade;
- quand le gaz présent dans la mine l'est à un niveau mortel, les mineurs risquent de mourir d'asphyxie;
- quand le niveau de gaz est très élevé dans la mine, une étincelle peut provoquer une explosion et tuer les mineurs présents.

Ces trois propriétés (inondation, suffocation, explosion) sont des obstacles au but principal. Ils peuvent être transformés an buts en utilisant le pattern « avoid ».

[17] Nous entendons dans le cadre de cette illustration la mine dans son ensemble (avec contrôle, pompe, alarme,…)

```
Goal Avoid[Drowning]
Goal Avoid[Suffocation]
Goal Avoid[Explosion]
```

Il est évident que d'autres problèmes peuvent survenir dans la mine, et donc constituer autant d'obstacles. De même, d'autres buts de plus haut niveau (rendre la mine productive par exemple) peuvent être énoncés. Nous nous limiterons dans le cadre de cette illustration aux obstacles définis ci-dessus, ainsi qu'aux buts en rapport avec l'énoncé, sans élargir sa portée.
Les buts définis ici seront spécifiés plus loin, lorsque nous voudrons les raffiner.

b. Objets concernés par les buts

Nous allons présenter ici les objets cités plus haut, ainsi que ceux qui vont servir à opérationnaliser les buts.

```
Entity Mine
   Has waterLevel : Float
   Has gasLevel : Float
   Has waterFlow : Float
   Has WMIN : Float
   Has WMAX : Float
   Has MMAX : Float

Agent Miner
   Attribute alive : Boolean
```

En lisant l'énoncé, d'autres objets sont trouvés: la pompe, les trois détecteurs (WD, PD et GD), l'alarme et le système de contrôle.

```
Entity WaterLevelDetector
   Has level : Float

Entity GasLevelDetector
   Has level : Float

Entity PumpFailureDetector
   Has failure : Boolean

Agent Pump
   Attribute active : Boolean
   Attribute outOfOrder : Boolean
   Attribute FLOW : Float

Agent Alarm
   Attribute active : Boolean
```

Le système de contrôle doit être explicitement défini. Les trois détecteurs lui fournissent les informations relevées (input), tandis que la pompe et l'alarme sont les dispositifs qui vont réguler physiquement la mine suite à ses ordres (output). Le système de contrôle peut donc être vu comme l'agrégation de détecteurs (3), d'un controlleur et de régulateurs (2).

```
Agent ControlSystem

Entity Controller
```

Nous pouvons « relier » des entités entre elles, en en formalisant certains faits. Ainsi, on peut noter que:

- les détecteurs observent la mine (relation 'Observing');
- les régulateurs contrôlent/régulent la mine (relation 'Controlling')
- le détecteur PD observe la pompe (relation 'Observing').

Deux agrégations peuvent être effectuées: nous pouvons « regrouper » les trois détecteurs en une entité « Detector » (détecteur); nous pouvons « regrouper » la pompe et l'alarme en une entité « Actuator » (régulateur). Ces regroupements peuvent être modélisés par une relation « PartOf » dans notre modèle KAOS (correspondant à une relation IsA).

A partir de ces objets et relations trouvées, nous pouvons dresser un modèle des objets pour la mine (figure 2.8).

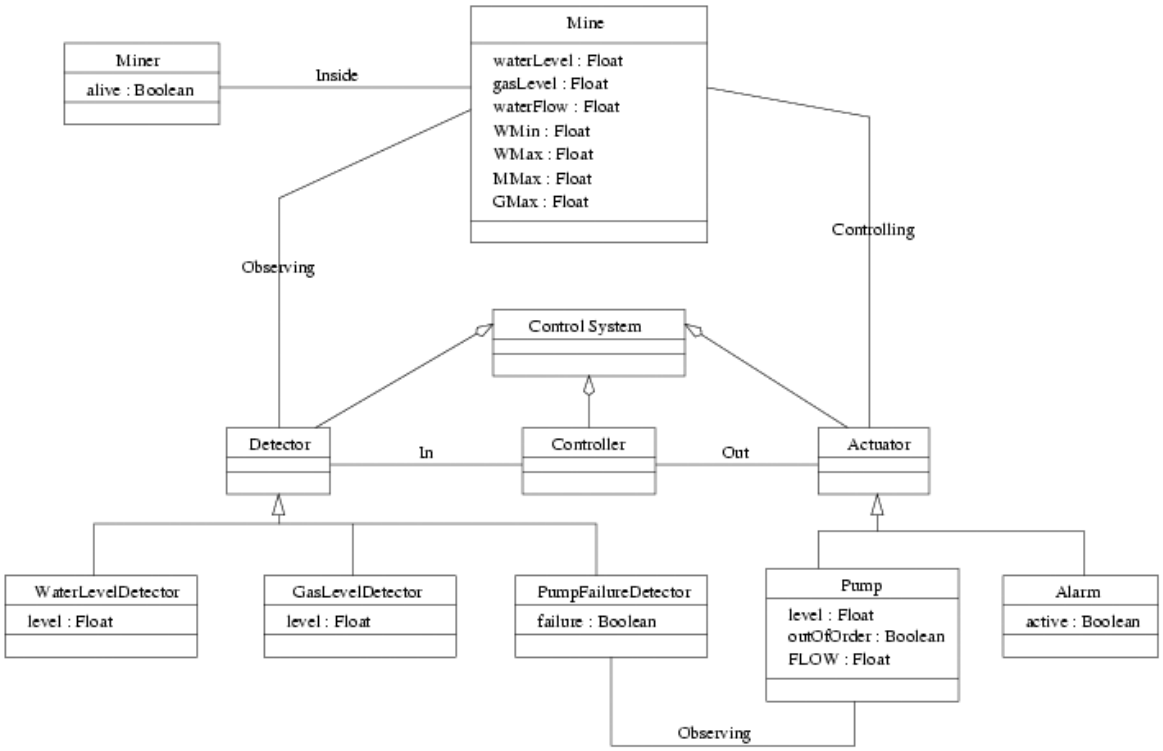


fig. 2.8 – modèle de la mine

Notons que :

- les attributs waterLevel, gasLevel de la mine sont des attributs dérivés du niveau supérieur (méta), et ne sont observables qu'à travers les WD et GD respectivement (via les attributs « level » de ces détecteurs, qui sont du niveau « domaine »)
- les attributs « active » de la pompe et l'alarme ne sont controllés que par le controlleur
- WMAX, WMIN, MMAX, GMAX et FLOW sont des constantes
- 0 < WMIN < WMAX
- les mineurs sont dans la mine (relation Inside)

(2) Identification préliminaire des agents potentiels du système auxquels ont peut associer des actions

c. Agents potentiels

Reprenons les agents identifiés à l'étape précédente:

- Le mineur: l'action « mourir » n'est pas prise en compte (ressusciter non plus). A travers la relation « Inside », nous voyons que le mineur peut entrer et sortir de la mine. L'évacuation doit également être retenue.
- La pompe: l'attribut « active » suggère que la pompe puisse être mise en marche et arrêtée (par le système de contrôle).
- L'alarme: même remarque que pour la pompe.

d. Actions

Nous avons donc identifié cinq actions, que nous allons définir. Les relations « Observing » et « Controlling » du modèle sont statiques, et ne sont utilisées que pour obtenir des références entre les instances. Il en va de même pour mesurer les niveaux (d'eau, de gaz), vider le réservoir et collecter l'eau, cités dans l'énoncé.

```
Action EnterMine
   In : Miner p, Mine m, Inside
   Out : Inside
   DomPre : ¬Inside(p,m)
   DomPost : Inside(p,m)
   DomTrig : start of working period
   CapableOf : Miner m

Action ExitMine
   In : Miner p, Mine m, Inside
   Out : Inside
   DomPre : Inside(p,m)
   DomPost : ¬Inside(p,m)
   DomTrig : end of working period
   CapableOf : Miner m

Action EvacuateMine
   In : Miner p, Mine m, Inside
   Out : Inside
   DomPre : true
   DomPost : (∀ p:Miner) : ¬Inside(p,m)
   CapableOf : Miner m
Action StartPump
   In : Pump p
   Out : Pump p
   DomPre : ¬p.active
   DomPost : p.active
   CapableOf : ControlSystem cs [, Miner m]
```

On pourrait supposer qu'un mineur soit superviseur et puisse activer la pompe ainsi que l'alarme. Nous ne considèrerons pas ce cas ici.

```
Action StopPump
   In : Pump p
   Out : Pump p
```

```
   DomPre : p.active
   DomPost : ¬p.active
   CapableOf : ControlSystem cs

Action StartAlarm
   In : Alarm a
   Out : Alarm a
   DomPre : ¬a.active
   DomPost : a.active
   CapableOf : ControlSystem cs

Action StopAlarm
   In : Alarm a
   Out : Alarm a
   DomPre : a.active
   DomPost : ¬a.active
   CapableOf : ControlSystem cs
```

(3) Opérationnalisation des buts en Exigences

Nous allons reprendre et traiter chacun des trois buts de haut niveau qui ont raffinés à partir de notre but de départ Maintain[SecureMine].

Avoid[DrowningDanger]

Ce but peut être spécifié de la manière suivante:

```
Goal Avoid[DrowningDanger]
   FormalDef :   (∀ p:Miner, m:Mine)
                 Inside(p,m) ⇒ □ ¬(m.level ≥ m.WMAX)
```

Ce but peut être décomposé en deux autres buts: éviter l'inondation dans la mine et évacuer la mine en cas d'inondation. Il s'agit ici d'un raffinement-ET.

```
Goal Avoid[Flooding]
   FormalDef : (∀ m:Mine) □ ¬(m.level ≥ m.WMAX)

Goal Achieve[EvacuationWhenFloodingDanger]
   FormalDef :   (∀ p:Miner, m:Mine)
                 (m.waterLevel m.WMAX) ⇒ ◊ ¬Inside(p,m)
```

Le premier de ces sous-buts peut être à son tour décomposé. On supposera ici que le niveau d'eau dans la mine est mesuré correctement (hypothèse). De plus, il faut s'assurer que le niveau est toujours contrôlé.

```
Assumption Maintain[AccurateWaterLevelMeasure]
   FormalDef : (∀ m:Mine, wd:WaterLevelDetector)
               Observing(wd,m) ⇒ □ (wd.level = m.waterLevel)

Goal Next[WaterLevelControlled]
   FormalDef :   (∀ m:Mine, wd:WaterLevelDetector, p:Pump)
                 Observing(wd,m) ∧
                 (wd.level = m.WMAX) ⇒ o (wd.level < m.WMAX)
```

Ceci peut être opérationnalisé. En effet, contrôler le niveau d'eau de la mine revient à activer la pompe et à la désactiver suivant le niveau d'eau observé. Deux hypothèses doivent sont émises: la pompe a un débit suffisant et elle fonctionne (n'est pas arrêtée).

Assumption Maintain[PumpHasSufficientFlow]
FormalDef : (∀ p:Pump) □ (p.FLOW > m.waterFlow)

Assumption Avoid[PumpOutOfOrder]
FormalDef : (∀ p:Pump) □ ¬p.outOfOrder

Venons-en maintenant au deuxième sous but (Goal Achieve[EvacuationWhenFloodingDanger]). L'évacuation sous-tend que l'on puisse activer une alarme pour prévenir les mineurs. On supposera également que les mineurs respecte l'alarme et sorte quand elle retentit.

Goal Achieve[AlarmTriggered]
FormalDef : (∀ m:Mine, wd:WaterLevelDetector, a:Alarm)
Observing(wd,m) ∧ (wd.level > m.MMAX) ∧
Controlling(a,m) ⇒ ◊ a.active

Assumption Achieve[MineEvacuatedOnAlarm]
FormalDef : (∀ m:Mine, p:Miner, a:Alarm)
Controlling(a,m) ∧ (a.active) ⇒ ◊ ¬Inside(p,m)

On peut également supposer que le niveau d'eau est mesuré correctement, si on veut évacuer pour des raisons valables (cette hypothèse a déjà été spécifiée).
Voici ce que donne notre décomposition pour le but Avoid[DrowningDanger][18]:

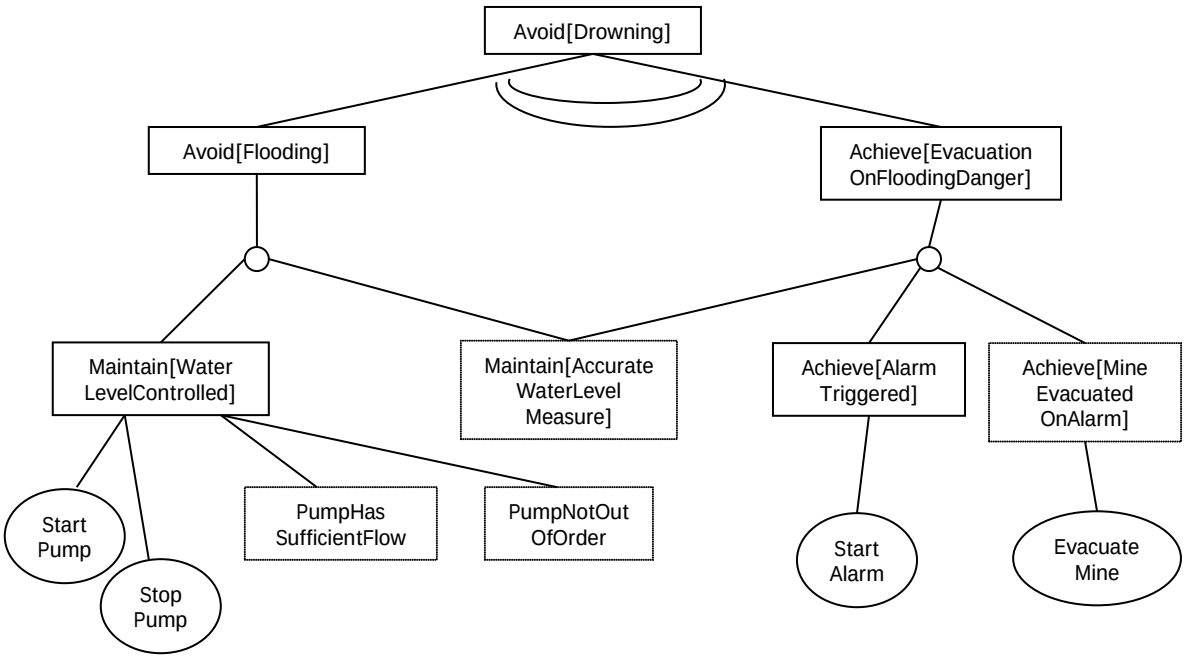


fig. 2.9 – arbre de décomposition

[18] La syntaxe graphique que nous utiliserons tout au long de ce mémoire est décrite en annexe

Ceci n'est pas suffisant. Le problème du bon fonctionnement de la pompe devrait être pris en compte plus haut dans le graphe. Ceci au niveau des buts Avoid[Flooding] et Achieve[EvacuationWhenFloodingDanger]. Nous pouvons les redéfinir comme suit:

Goal Achieve[NoFloodinfWhenPumpNotOutOfOrder]
FormalDef : (∀ m:Mine, p:Pump)
¬(Controlling(p,m) ∧ p.outOfOrder) ⇒ ◊ (m.level < m.WMAX)

Goal Achieve[EvacuationOnFloodingDangerAndPumpNotOutOfOrder]
FormalDef : (∀ m:Mine, p:Pump, n:Miner)
(Controlling(p,m) ∧ p.outOfOrder) ⇒ ◊ (¬Inside(n,m))

Le but Achieve[AlarmTriggered] est également redéfini, pour les mêmes raisons que celles citées ci-dessus:

Goal Achieve[AlarmTriggered]
FormalDef :
(∀ m:Mine, wd:WaterLevelDetector, a:Alarm, p: Pump,
dp:PumpFailureDetector)
Observing(wd,m) ∧ Controlling(a,m) ∧ Observing(dp,p) ∧
(wd.level > m.MMAX) ∧ ¬dp.failure ⇒ ◊ a.active

Une hypothèse supplémentaire est émise: l'état de la pompe est sous surveillance:

Assumption Maintain[PumpStateObserved]
FormalDef : (∀ p:Pump, dp:PumpFailureDetector)
Observing(dp,p) ⇒ □ (dp.failure = p.outOfOrder)

Le nouveau graphe décompose Avoid[Drowning] avec un nœud OU au lieu d'un nœud ET, et l'hypothèse Maintain[PumpnotOutOfOrder] (décomposant Maintain [WaterLevelControlled]) ne sert plus à rien.

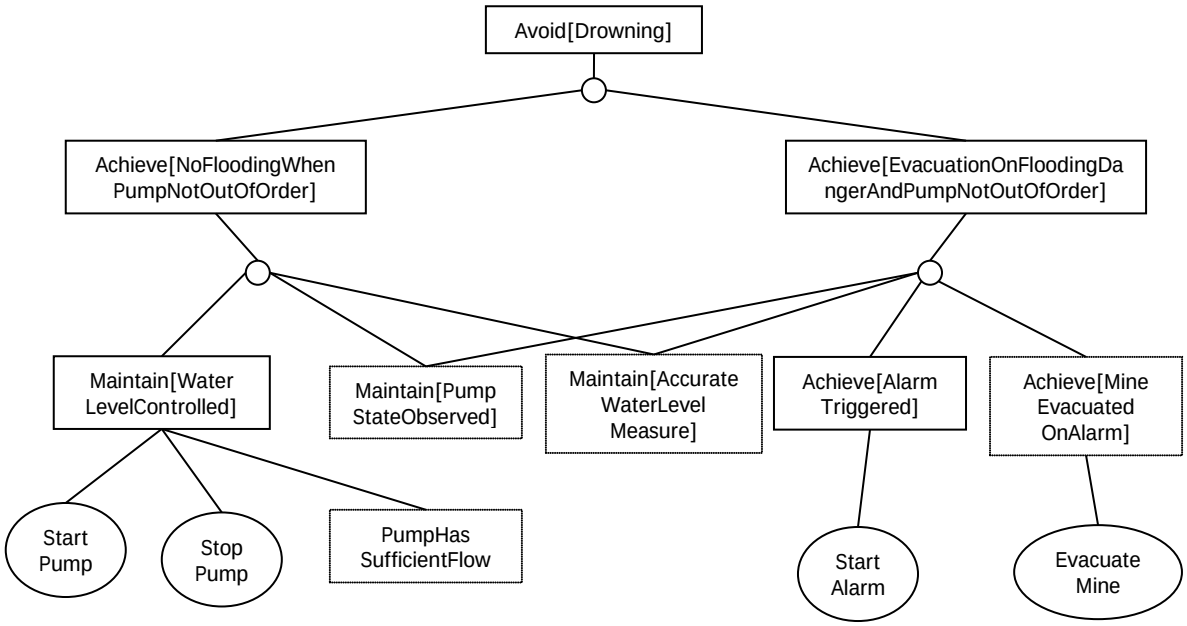


fig. 2.10 – arbre de décomposition

Avoid[Suffocation]

Spécifions le but:

Goal Avoid[Suffocation]
FormalDef : (∀ m:Mine, p:Miner)
Inside(p,m) ⇒ □ ¬Suffocate(p)

Eviter la suffocation revient à évacuer la mine lorsque le niveau de gaz est trop élevé. Ce sous-but, seul raffinement du but supéreieur, se décompose de manière très similaire à celle du but [EvacuationWhenFloodingDanger]. On y retrouve les hypothèses concernant la mesure correcte du niveau (de gaz) mesuré et l'évacuation correcte de la mine. Le but Achieve[AlarmTriggered] est aussi une des composantes du raffinement.

Goal Achieve[EvacuationWhenSuffocationDanger]
FormalDef : (∀ p:Miner, m:Mine)
(m.gasLevel > m.GMAX) ⇒ ◊ ¬Inside(p,m)

Assumption Maintain[AccurateGasLevelMeasure]
FormalDef : (∀ m:Mine, gd:GasLevelDetector)
Observing(gd,m) ⇒ □ (gd.level = m.gasLevel)

Goal Achieve[AlarmTriggered]
FormalDef : (∀ m:Mine, gd:GasLevelDetector, a:Alarm)
Observing(gd,m) ∧ (gd.level > m.GMAX) ∧
Controlling(a,m) ⇒ ◊ a.active

Assumption Achieve[MineEvacuatedOnAlarm]
FormalDef : (∀ m:Mine, p:Miner, a:Alarm)
Controlling(a,m) ∧ (a.active)
⇒ ◊ ¬Inside(p,m)

Voici donc ce que nous obtenons:

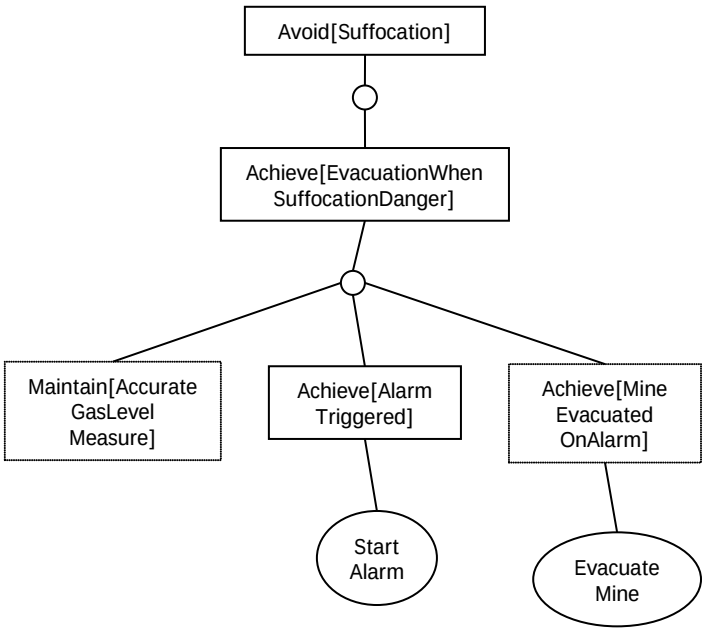


fig. 2.11 – arbre de décomposition

Avoid[Explosion]

Spécifions le but:

```
Goal Avoid[Explosion]
   FormalDef : (∀ m:Mine) ◊ ¬Explode(m)
```

En fait, ceci revient à éviter de faire des étincelles quand le niveau de gaz est trop élevé, ce qui nous ramène à des concepts que nous connaissons.

```
Goal Avoid[SparksWhenHighGasConcentration]
   FormalDef :   (∀ m:Mine)
                 (m.gasLevel ≥ GMAX) ⇒ □¬ MakingSparks(m)
```

Considérons ce qui peut provoquer des étincelles: feu, cigarettes, allumettes, interrupteurs (lampes), moteurs,… Nous ne considérerons ici que le cas du moteur. Et il y en a un dans la mine, c'est la pompe. Il faudra donc la couper lorsque le niveau de gaz est trop élevé dans la mine.

```
Goal Achieve[PumpOffWhenHighGasConcentration]
   FormalDef :   (∀ m:Mine, p:Pump)
                 (m.gasLevel > m.GMAX) ∧ Controlling(p,m)
                 ⇒ ◊ ¬p.active
```

L'hypothèse est de nouveau faite que le niveau de gaz mesuré est correct. La contrainte suivante sera opérationnalisé par le système : couper la pompe quand le niveau de gaz est trop élevé.

```
Assumption Maintain[AccurateGasLevelMeasure]
   FormalDef :   (∀ m:Mine, gd:GasLevelDetector)
                 Observing(gd,m) ⇒ □ (gd.level = m.gasLevel)

Goal Achieve[PumpTurnedDown]
   FormalDef :   (∀ m:Mine, gd:GasLevelDetector, p:Pump)
                 Observing(gd,m) ∧ Controlling(p,m) ∧
                 (gd.level > m.GMAX) ⇒ ◊ ¬p.active
```

Voici ce que nous obtenons:

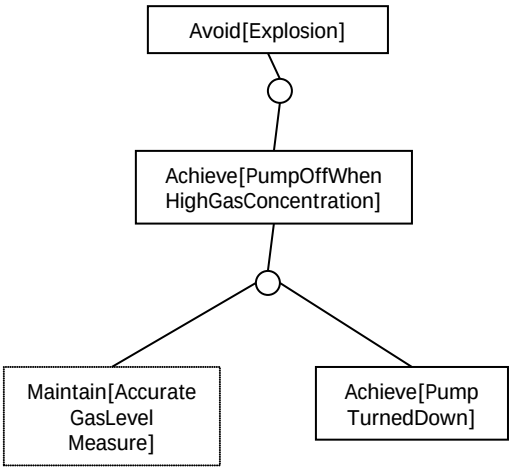


fig. 2.12 – arbre de décomposition

(4) Raffinement des objets et actions

Les raffinement des actions ont déjà été fait lors du point (2). En ce qui concerne les objets, il n'y a pas à raffiner, le problème étant assez simpliste.

(5) Dérivation des conditions portant sur les objets et actions pour prendre en compte les contraintes

La seule condition portant sur des objets (en l'occurrence, des agents) est Inside, elle n'a pas besoin d'être dérivée.

Les actions quant à elles, peuvent être spécifiées plus précisément, maintenant que la décomposition des buts est claire et précise.

(6) Identification des responsabilités supplémentaires des agents

Pas de responsabilité supplémentaire dans le cadre de cet exemple. On a déjà parlé du fait qu'un mineur pourrait déclencher l'alarme, on pourrait également supposer que ce soit un mineur (superviseur ou responsable sécurité) qui coupe l'alarme et autorise les autres à revenir dans la mine.

(7) Assignation des actions aux agents, qui vont en devenir responsables

Ces assignations ont été faites au point (2).

Nous voyons, si nous regroupons les trois graphiques de décomposition en sous-buts, que deux buts entrent en conflit (voir figure ci-dessous). En effet, pour contrôler le niveau d'eau, il faut utiliser la pompe, ce qui est impossible si le niveau de gaz est trop élevé.

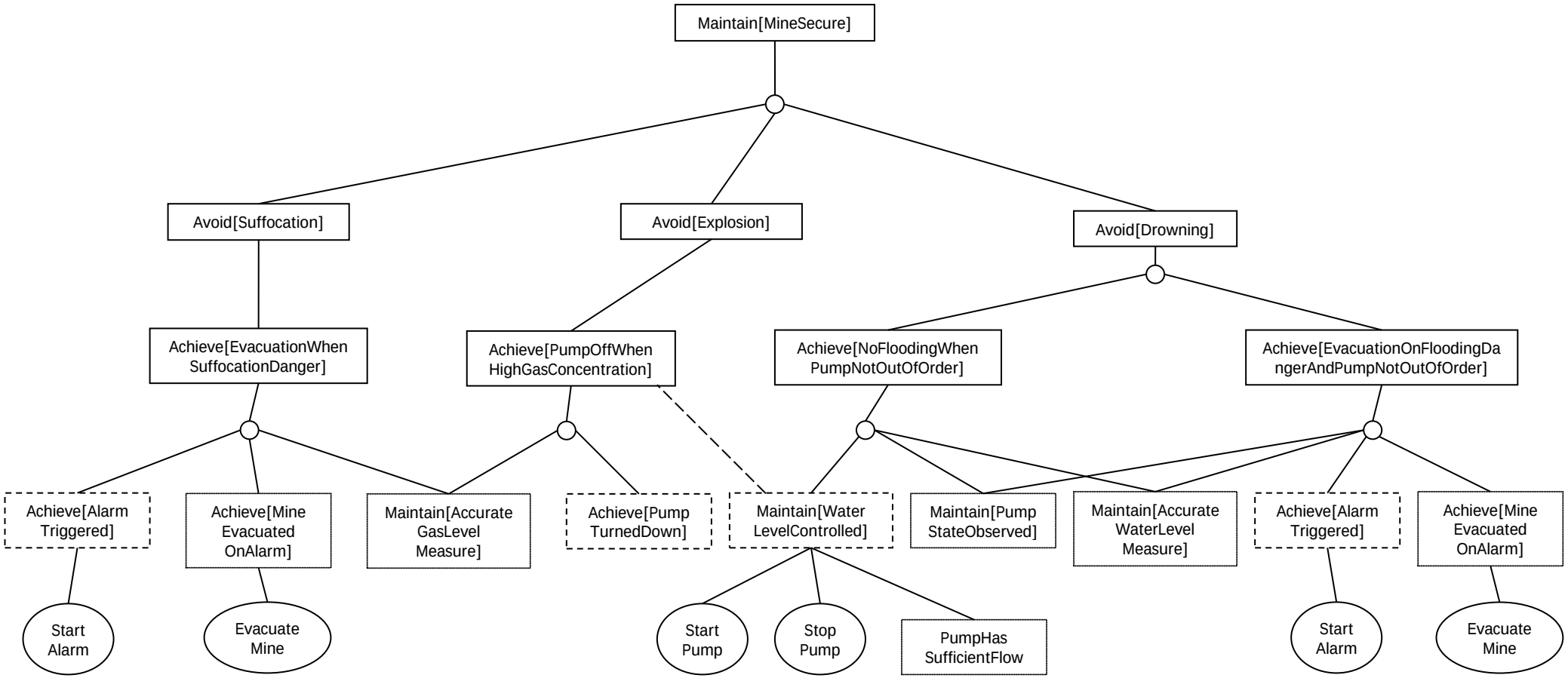


fig. 2.13 – arbre de décomposition

2.6. Enseignements tirés de l'étude de KAOS

La méthode KAOS nous a permis d'identifier les buts que le système doit atteindre. Nous avons pu décomposer ses buts afin d'obtenir des exigences assignables à des agents identifiés précédemment.

Comme nous l'avons remarqué au début de ce chapitre, le modèle obtenu ci-dessus est peut-être beaucoup plus proche de la réalité, mais encore incomplet. Il serait intéressant de pouvoir le vérifier en confrontant ce modèle avec le système existant. Notre travail nous a permis de le faire, et pouvoir obtenir un « feedback » qui permet d'améliorer le modèle KAOS. A cette fin, nous nous sommes servis d'ASAX pour surveiller le système avec les exigences du modèle comme base (les seuls buts assignables à des agents, dont l'exécution des actions associées peuvent être identifiées dans une trace des événements du système). Nous allons maintenant présenter ASAX et l'intérêt que ce logiciel représente dans le cadre de la « vérification » du modèle KAOS.

3. ASAX

Asax est le composant « background » que nous utilisons dans le cadre de ce mémoire. Il a pour but d'analyser les traces d'exécution, au moyen de règles RUSSEL dérivées des règles exprimées en logique temporelle. Il faut cependant bien faire la distinction entre *la vérification / l'analyse* des traces d'exécution, nécessitant une *traduction* des règles, sous la « responsabilité » de Asax, et la *spécification* du comportement, nécessitant la *création* des règles, sous la « responsabilité » de Kaos. Nous reviendrons en détail sur la répartition des différentes parties du travail dans le chapitre 5.

3.1. Principes généraux d'ASAX

Les principes à la base de l' « Advanced Sequential file Analysis on uniX » sont assez simples. Il s'agit d'un outil offrant un ensemble de mécanismes d'analyse de fichiers séquentiels. Les maîtres mots de sa conception sont efficacité et généricité.

Les types de traitements réalisables sur les fichiers séquentiels ne sont a priori limités que par l'imagination de l'utilisateur. On peut, notamment :

- Sélectionner des enregistrements (records) satisfaisant à une condition plus ou moins élaborée, les imprimer, les stocker…
- Sélectionner des séquences d'enregistrements satisfaisant à des conditions et / ou des relations spécifiques « inter-records ».
- Déterminer des comportements du système modélisé par son « fichier de trace », à partir des séquences de records contenues dans ladite trace.
- Surveiller lesdits comportements et générer des alarmes et / ou messages lorsque les comportements satisfont ou ne satisfont pas à des standards préétablis.
- Générer des statistiques dérivées du comportement d'un système ou de ses utilisateurs.
- …

3.1.1. Généricité

Asax est un outil générique, en ce sens qu'il est capable d'analyser n'importe quel type de fichier séquentiel. Cette fonctionnalité est rendue possible par un mécanisme de traduction systématique du fichier séquentiel d'entrée (trace). Ce dernier est traduit dans un format dit NADF, pour Normalized Audit Data Format. C'est ce fichier NADF qui sera effectivement analysé par Asax. Le format NADF sera détaillé plus loin dans ce chapitre.

3.1.2. Efficacité

Deux principes-clé d'ASAX permettent d'obtenir un degré d'efficacité nécessaire à l'analyse de fichiers pouvant être, le plus souvent, gigantesques. Ces principes sont :

- L'analyse en une passe : Le langage RUSSEL utilisé dans ASAX permet des analyses efficaces de grands fichiers en une et une seule passe. Il n'y a pas de « backtracking »[19].

- Les étapes répétitives sont optimisées via des techniques efficientes d'implémentation. On notera par exemple l'utilisation systématique de « piles » de règles[20].

3.1.3. Puissance

La puissance de traitement de Asax est contenue dans son langage de définition de règles, RUSSEL (Rule-baSed Sequential Evaluation Language), qui permet d'exprimer des règles allant de la simple sélection à l'expression d'assertions logiques complexes… Une section est réservée plus loin à la présentation de RUSSEL.

3.1.4. Portabilité

Conséquence directe de sa généricité, Asax est facilement portable et utilisable sur différentes architectures (munies d'un compilateur C), et pour différents usages. Asax n'est aucunement lié à un système d'exploitation ou à une architecture particulière, même si certains de ses composants (adaptateur de format…) le sont.

3.1.5. Extensibilité

Le langage RUSSEL a été conçu pour offrir une interface avec le langage C. On peut facilement définir de nouveaux outils d'analyse sous la forme de routines C, et les intégrer par la suite dans Asax.

[19] Bien qu'il soit possible, afin de contourner ce problème, de, par exemple, générer une trace inversée et de relancer ASAX sur cette trace

[20] Le terme de pile, cité ici, est à prendre avec précaution. En effet, l'ordre d'exécution des règles par ASAX n'est pas précisé.

3.2. Architecture d'ASAX

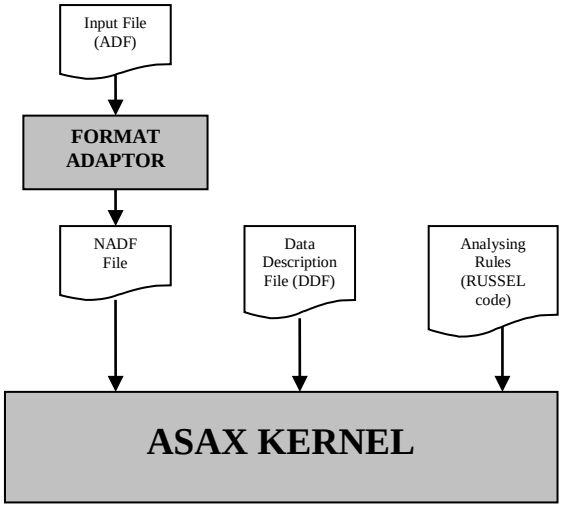


fig. 3.1 – architecture asax

Détaillons les différents composants :

3.2.1. La trace

C'est, concrètement, le fichier d'entrée à analyser. Il peut être de n'importe quel type : fichier d'audit, trace d'exécution d'un programme, trafic réseau…

A chaque type de fichier d'entrée peut correspondre un format, a priori incompatible avec celui compréhensible par Asax. Il est donc nécessaire de décoder / traduire ce fichier d'entrée pour l'adapter au format NADF.

3.2.2. Adaptateur de format

L'adaptateur de format est un composant C intégré à ASAX (routine « ConvBSM »), mais aussi un exécutable séparé (« FormatAdaptor »), utilisé par ASAX dasn la détection d'intrusions sous Solaris/Sparc ou Intel.

Comme nous l'avons vu plus haut, ASAX est *générique* et *extensible*. L'adaptateur de format illustre bien ces particularités. En tant que routine C, il peut être intégré à ASAX, mais pas au même titre que n'importe quelle nouvelle fonctionnalité[21]. Cela permet une utilisation transparente pour l'utilisateur.

En tant que traducteur, l'adaptateur de format permet la généricité d'Asax face à la multitude de formats d'entrée possibles. Un adaptateur de format sera donc développé pour chaque nouveau type de format d'entrée. L'utilisateur désire changer de type de fichier d'entrée, il n'a qu'à faire appel à un nouvel adaptateur de format…

[21] Les autres fonctionnalités sont en fait ajoutées au langage RUSSEL (voir 3.2.5)

3.2.3. Le Fichier NADF

C'est, comme nous l'avons dit plus haut, le fichier qui sera analysé concrètement par Asax. Ce n'est pas tellement le fichier en tant que tel qui nous intéresse, mais plutôt son format. Détaillons-le :

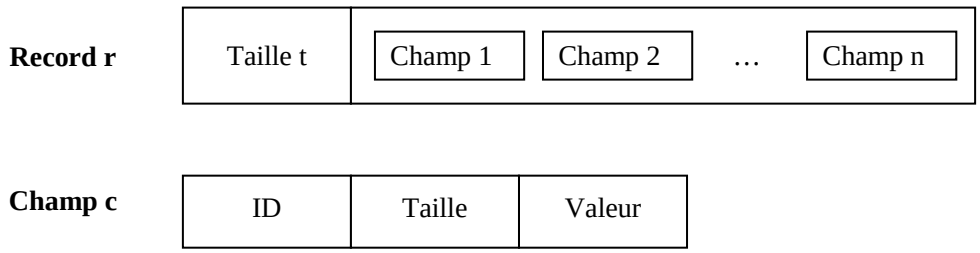


fig. 3.2 – format NADF

- chaque record contient :
    - La taille réelle du record, comprenant la taille du contenu (les différents champs) à laquelle on ajoute la taille du champ contenant ladite taille (généralement 4 octets).
    - Un nombre variable de champs, contenant l'information à analyser.

- chaque champ contient :
    - Un identifiant, repris sous la forme d'un entier codé sur 2 octets. On verra plus loin comment désigner un champ, soit par son identifiant, soit par un nom, ce qui est plus lisible.
    - Une longueur, codée également sur 2 octets, reprenant uniquement la taille du champ « Valeur », en excluant la taille des deux champs « ID » et « Taille ».
    - Le champ « Valeur » reprenant, comme son nom l'indique, la donnée à analyser.

Notons également que :

- Le codage des différents champs est dépendant de la machine sur laquelle il est généré. Nous évoquerons donc les architectures BigEndian (sur Sparc) et LittleEndian (sur Intel x86) qui influent sur l'ordre des bits/octets.
- Dans un record donné, les champs commencent toujours à une adresse paire.
- Même si, pour Asax, le caractère séquentiel est le seul requis, il est souvent nécessaire, voire indispensable, de respecter un caractère chronologique dans le fichier.

3.2.4. Le descripteur de format

Matérialisé par un fichier DDF (Data Description File), fichier texte séquentiel reprenant la description des types d'informations contenues dans les champs du fichier NADF. La syntaxe concrète du descripteur de format peut se consulter en annexe. En voici cependant les grandes lignes.
Pour décrire une donnée, on fournit les différentes informations suivantes :

- L'identifiant (entier sur 2 octets)
- Le type d'origine
- Le type de destination
- Le nom du champ, permettant une désignation plus intuitive en RUSSEL (par rapport à la définition par numéro identifiant)
- La description sémantique

Chaque information est reprise sur une et une seule ligne, préfixée de 1 à 5. Au chapitre des particularités, notons que le type d'origine désigne comment était considérée la donnée dans le fichier de trace, alors que le type de destination désigne le type de la donnée au sein du fichier NADF. Notons également, comme on peut s'en douter, que l'identifiant repris dans le fichier DDF est celui du champ ID de la donnée correspondante du fichier NADF.

Les six lignes d'en-tête sont optionnelles. Elles sont cependant souvent utiles pour savoir sur quelle architecture a été générée le fichier NADF, et donc savoir quel type de codage a été utilisé (BigEndian, LittleEndian…). On y retrouve, préfixé de A à F :

- Le nom et la version du software générant les informations
- La machine sur laquelle a été générée l'information
- La date de création
- La version du programme produisant le fichier DDF
- La machine sur laquelle a été généré le fichier DDF
- La date de création du fichier DDF

3.2.5. Le programme RUSSEL

Ici encore, ce n'est pas tant le programme en lui-même qui nous intéresse, mais plutôt le langage de programmation RUSSEL, sa syntaxe, sa sémantique, sa structure…

Précisons d'emblée que le lecteur pourra consulter en annexe une version exhaustive de la syntaxe RUSSEL. Nous nous bornerons ici à en définir les concepts essentiels.

a. Les concepts de base :

- Le mode de fonctionnement :

Comme nous l'avons déjà évoqué, RUSSEL est un langage « orienté règles ». L'utilisateur défini un ensemble de règles de traitement, sélection, action… à

appliquer aux différents records. Ces règles sont soit *actives*, soit *inactives*. Une règle active est une règle qui doit s'appliquer au record courant. L'exécution d'un programme RUSSEL peut donc être vue comme l'application d'un ensemble de règles au record courrant. Une fois que toutes les règles de cet ensemble sont appliquées, on passe simplement au record suivant. Rappelons que le backtracking n'est pas utilisé.

- L'environnement :

Par environnement courant, nous désignons :

a. Le record courant.
b. L'environnement local, soit l'ensemble des variables représentant la mémoire de travail du système. En l'absence de backtracking, c'est la seule mémoire disponible.
c. La pile des règles courantes. C'est l'ensemble, a priori non ordonné, des règles actives pour le record courant.
d. La pile des règles suivantes. C'est l'ensemble des règles actives pour le record suivant. Cette pile sera bien entendu « recopiée » dans la précédente à chaque itération dans le fichier.
e. La pile des règles de clôture. C'est l'ensemble des règles à appliquer / activer après le traitement du dernier record.

Schématiquement, ceci donne :

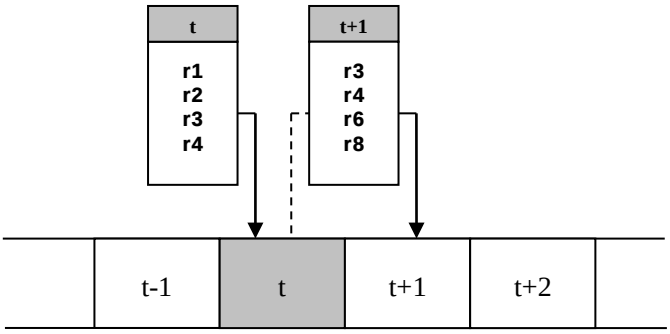


fig. 3.3 –piles des règles

La construction des différentes piles est dynamique, en ce sens qu'elle est construite au fur et à mesure de l'exécution du programme. On peut donc, activer des règles « for current », « for next » ou « at completion ». L'ordre d'exécution des règles n'est pas défini. Nous avons pourtant constaté, à l'usage que la pile est organisée, sur une architecture monoprocesseur, en « last-in first-out ».

b. Les types de données :

Les seuls types de données reconnus par RUSSEL sont les entiers et les chaînes d'octets bruts. Les opérations courantes (comparaison, égalité, opérations arithmétiques sur les entiers) sur ces types sont supportées. La conversion de types est également supportée dans le sens string → entier. Elle est assurée par la routine « strToInt ». Les autres opérations éventuelles sont à développer via des routines C en vue d'une intégration à Asax.

c. Les variables :

Les variables sont soit globales soit locales. Une variable globale a comme portée soit le module dans lequel elle est définie (si elle est déclarée « internal ») soit le programme complet (si elle est déclarée « external »). La variable locale est déclarée à l'intérieur d'une règle. Sa portée se limite à ladite règle.

d. Les expressions :

Une version plus exhaustive des syntaxe et sémantique des expressions se trouve en annexe. Nous nous bornerons ici à citer :

a. La constante.
b. Le nom de champ.
c. Les expressions arithmétiques.
d. L'appelle de routines.
e. Les conditions / comparaisons.

e. Les instructions :

Ici aussi, nous renvoyons le lecteur aux annexes pour trouver une version complète. Quelques mots d'explications cependant concernant les instructions définies en RUSSEL :

a. Skip : Désigne l'instruction vide.
b. L'assignation : Très classiquement, se présente sous la forme valGauche := valDroite.
c. L'instruction conditionnelle : Elle a une forme un peu particulière. L'instruction conditionnelle est présentée sous la forme :

```
if
  cond₁ → action₁
  cond₂ → action₂
  …
  condₙ → actionₙ
fi
```

et est exécutée comme suit : si l'évaluation de la 1ère condition renvoie vrai, on effectue la 1ère action puis on sort de l'instruction. Sinon, on évalue la condition suivante. Dans cette optique, la très classique forme « if-then-else » trouve son équivalent dans :

```
if
  cond₁ → action₁
  true  → action₂
fi
```

d. L'instruction répétitive : Elle se présente sous une forme similaire à la conditionnelle, à ceci près qu'on ne sort de l'instruction que lorsque toutes les conditions sont évaluées à faux.

e. Le déclenchement de règles : Sans doute la principale fonctionnalité de RUSSEL. Elle se présente sous la forme « trigger off <mode de déclenchement> <nom de la règle> », où le mode de déclenchement est soit :

for_current :	la règle est ajoutée à la pile des règles actives pour le record courant
for_next :	la règle est ajoutée à la pile des règles actives pour le record suivant
at_completion :	La règle est ajoutée à la pile des règles actives pour la fin du traitement (après traitement du dernier record).

f. Les règles :

C'est ici que, d'une part, l'on définit les règles (nom, attributs…) et que, d'autre part, on décrit leur fonctionnement. Ce fonctionnement peut, typiquement, être la génération d'un message ou l'activation d'une ou plusieurs autres règles. La « première règle » est définie comme étant l' « init_action » qui décrit toutes les opérations à effectuer avant le traitement du 1er record. C'est elle qui, entre-autre, initialisera les piles.

3.3. Illustration

Pour illustrer nos propos, voici un petit programme RUSSEL. Celui-ci compte le nombre de mauvaises connexions d'utilisateurs sur une machine donnée. Le code indiqué ci-dessous a été simplifié afin d'en améliorer la clarté.

```
global internal count : integer;

init_action;
begin
   count := 0;
   trigger off for_next count_bad_login;
   trigger off at_completion print_bad_login
end;

rule count_bad_login;
begin
   if EVENT = 'login' and RESULT = 'failure' --> count := count + 1
   fi;
   trigger off for_next count_bad_login
end;

rule print_bad_login;
begin
   println ('Number of bad logins detected : ', count)
end.
```

La première règle activée est toujours `init_action` ; à ce moment, aucun record de la trace n'a encore été traité. `init_action` va activer les règles pour le premier record de la trace et/ou pour la terminaison du traitement de celle-ci. La règle `count_bad_login` est activée pour chaque record de la trace et compte le nombre de mauvaises connexions (en utilisant la variable globale `count`). Quand la fin de la trace est atteinte (`at_completion`), le nombre de mauvaises connexions est affiché à l'écran.

Des exemples plus concrets d'applications d'ASAX en sécurité peuvent être trouvés sur les sites des cours de sécurité de systèmes d'exploitation données aux FUNDP (http://www.info.fundp.ac.be/ cgi-bin/ids) et à l'UCL (http://www.info.ucl.ac.be/notes_de_cours/Ingi2591/).

4. Logique temporelle

Une formule de logique temporelle est construite à partir d'une formule de logique classique, à laquelle nous appliquons des opérateurs temporels, des connecteurs logiques et des quantifications.

Nous présentons donc ci-dessous, non seulement la sémantique d'un sous-ensemble de la logique temporelle, mais aussi le sous-ensemble de la logique classique sur laquelle elle se fonde.

Nous reviendrons également sur les concepts évoqués dans l' « introduction à la logique temporelle », principalement le concept d'histoire, et les différents patterns utilisés.

Comme nous l'avons évoqué lors de l'introduction à la logique temporelle, nous passons ici de la compréhension « intuitive » à une analyse et une compréhension en profondeur des différents aspects de la logique temporelle utilisés dans le cadre de ce mémoire. De plus, à la fin de ce chapitre, nous fournirons un ensemble de fonctions CaML. Ces fonctions nous serviront par la suite, lors de la traduction des patterns de logique temporelle en programme RUSSEL. Cette traduction se déroulera en effet en 4 phases : la définition de la sémantique des différents patterns (dans ce chapitre), la définition des fonctions CaML et leur vérification (dans ce chapitre également), ce qui nous permettra d'avoir une expression des différents patterns plus facilement manipulables et, à notre avis, plus proche d'une description « opérationnelle ». Ensuite, les deux dernières phases, présentées dans le chapitre suivant, sont d'une part la version que nous pouvons appeler « opérationnelle », sous forme de diagrammes, où l'on modifie les fonctions CaML pour répondre à certaines contraintes opérationnelles (format de la trace des évènements, découpe en « tranches de temps »…) et la traduction proprement dite en RUSSEL.

4.1. Domaines sémantiques

Ici sont repris les différents domaines de définition des objets que nous utiliserons. Par « objets », nous désignons ici les différents concepts manipulables dans une analyse sémantique. Ils peuvent être aussi simple que des booléens, aussi complexes que des formules de logique temporelle.

Ces domaines sémantiques sont donc en quelque sorte les « types abstraits » du langage. Leur forme « concrète » sera définie plus tard. Nous n'en avons pas encore besoin à ce stade.

Les différents domaines sont :

$b \in \mathbb{B}$: Booléens

Correspondant à la définition classique du booléen. Peut donc prendre des valeurs vraie ou fausse.

i ∈ I : Entiers

Correspond également à une définition classique d'entiers, signés, sans borne supérieure ni inférieure.

sv ∈ Sv : Valeurs Simples

Union disjointe des deux domaines précédents, définit en fait le domaine de destination d'une évaluation d'expression.

v ∈ V : Vocabulaire

Définit l'ensemble des identificateurs, dénotant des variables des formules considérées. Un élément du vocabulaire (un identificateur) défini de façon univoque une variable relative à une formule considérée.

s ∈ S : Etat

Définit une « valuation », sous la forme d'une fonction définie comme suit :

$$V \rightarrow SV$$

Qui relie donc un ensemble d'identifiants ($\in$ **V**) aux valeurs des variables qu'ils dénotent ($\in$ **SV**).

m ∈ M : Modèle

Le modèle est défini comme une liste indexée d'états. Il est équivalent au concept d'histoire défini dans l'introduction. Chaque élément du modèle est donc une valuation, « identifiée » par son index entier.

$$(S * I)\ \mathtt{List}$$

exp ∈ Exp : Expressions

Représente les expressions définies sur le domaine V, c'est à dire les expressions qui ont pour variables un sous-ensemble des variables de V. Il s'agit donc des constantes, des variables proprement dites, et des expressions définies grâce aux connecteurs logiques ($\wedge$, $\vee$, $\neg$), ou entiers (+, -).

af ∈ Af : Formules atomiques

Une formule atomique est soit un atome (formule d'état) soit un prédicat. Les « opérateurs de prédicat » sont <, >, =.

bf ∈ BF : Formules booléennes

Dénote les formules construites à partir des connecteurs booléens ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$).

sf ∈ SF : Formules d'état

Dénote les formules de logique classique, quantifiées existentiellement (∃) ou universellement (∀), construites à partir de connecteurs booléens (∧, ∨, ¬, ⇒, ⇔).

tf ∈ TF : Formules temporelles

Dénote les formules de logique temporelle, avec leurs constructions à l'aide de connecteurs temporels (o, □, ◊, *U*, *W*,…).

4.2. Syntaxe abstraite

Voici une définition plus formelle des objets utilisés dans ce chapitre. Rappelons qu'il ne s'agit pas encore de la forme concrète des objets, qui sera définie par après.

```
B    ::= Vrai | Faux
I    ::= 1 | 2 | 3 | …
V    ::= a | b | c | …

opu  ::= - | ¬
```

Définition des opérateurs unaires

```
opb  ::= + | - | ∧ | ∨
```

Définition des opérateurs binaires

```
opp  ::= < | > | =
```

Définition des opérateurs de prédicats

```
opt  ::= □ | ~ | □ | • | □ | □ | •̃ | ◊̂ | ~̂ | ◊̂ | •̂ | ^□ | ♦̂
```

Définition des opérateurs temporels unaires

```
optb ::= U | W | B | S | Û | Ŵ | B̂ | Ŝ
```

Définition des opérateurs temporels binaires

```
exp ::=  V(x) | E(opu, exp) | E₂(opb, exp₁, exp₂) | C(x)
```

Définiition des expressions : soit une variable, soit un opérateur unaire appliqué à une expression, soit un opérateur binaire appliqué à deux expressions, soit une constante.

```
af  ::= Atom(x) | P(opp, exp*)
```

Définition des formules atomiques : soit un atome, soit un prédicat défini sur une liste d'expressions.

```
bf  ::= B(af) | NOT(bf) | OR(bf₁, bf₂) | AND(bf₁, bf₂) |
        IMPL(bf₁, bf₂) | EQUIV(bf₁, bf₂)
```

Définition des formules booléenes : soit une formule atomique, soit un opérateur booléen appliqué à une ou plusieurs formule(s) booléenne(s).

```
sf  ::= Assert(bf) | Exi(x, sf) | Uni(x, sf) | NOT(sf) |
        OR(sf₁, sf₂) | AND(sf₁, sf₂) | IMPL(sf₁, sf₂) |
        EQUIV(sf₁, sf₂)
```

Définition des formules d'état : soit une formule booléenne, soit une formule d'état quantifiée, soit un opérateur booléen appliqué à une ou plusieurs formule(s) d'état.

```
tf  ::= Tassert(sf) | Exi(x, tf) | Uni(x, tf) | NOT(tf) |
        OR(tf₁, tf₂) | AND(tf₁, tf₂) | IMPL(tf₁, tf₂) |
        EQUIV(tf₁, tf₂) | TOP₁(opt, tf) | TOP₂(optb, tf₁, tf₂)
```

Définition des formules temporelles : soit une formule d'état, soit une formule temporelle quantifiée, soit un opérateur booléen appliqué à une ou plusieurs formule(s) temporelle(s), soit un opérateur temporel appliqué à une ou plusieurs formule(s) temporelle(s).

4.3. Sémantique

Comme nous l'avons évoqué plus haut, il est avant tout nécessaire de présenter la sémantique de la logique classique sur laquelle se base la logique temporelle. Nous pourrons alors examiner les particularités propres aux opérateurs de logique temporelle, passés et futurs, ainsi que leurs versions strictes.

4.3.1. Sémantique des formules d'état

a. Les expressions :

Soit la fonction sémantique suivante :

$\mathcal{E}$: **exp → s → sv**

qui, à une expression associée à un état, renvoie la valeur correspondante de cette expression.

$\mathcal{E}$⟦V(x)⟧ s = s(x)
$\mathcal{E}$⟦C(x)⟧ s = x
$\mathcal{E}$⟦E(opu, exp)⟧ s = opu(e)
où e = $\mathcal{E}$⟦exp⟧ s
$\mathcal{E}$⟦E_2(opb, exp_1, exp_2)⟧ s = (e_1) opb (e_2)
où e_i = $\mathcal{E}$⟦exp_i⟧ s

Et, pour les listes d'expressions :

$\mathcal{E}^*$: **exp** List → **s** → **sv** List

$\mathcal{E}^*$ ⟦[]⟧ s = []
$\mathcal{E}^*$ ⟦head :: tail⟧ s = ($\mathcal{E}$ ⟦head⟧ s) :: $\mathcal{E}^*$ ⟦tail⟧ s

b. Les formules atomiques :

Soit la fonction sémantique suivante :

$\mathcal{A}$: **Af** → **S** → **B**

qui, à une formule atomique associée à un état, renvoie la valeur de vérité (booléenne) de la formule considérée.

$\mathcal{A}$ ⟦Atom(x)⟧ s = s(x)
$\mathcal{A}$ ⟦P(opp, explist)⟧ s = opp(e)
où e = $\mathcal{E}^*$ ⟦explist⟧ s

c. Les formules booléennes :

Soit la fonction sémantique suivante :

$\mathcal{B}$: **Bf** → **S** → **B**

qui, à une formule booléenne associée à un état, renvoie la valeur de vérité (booléenne) de la formule considérée.

$\mathcal{B}$ ⟦B(af)⟧ s = $\mathcal{A}$ ⟦af⟧ s
$\mathcal{B}$ ⟦NOT(bf)⟧ s = ¬ b
où b = $\mathcal{B}$ ⟦bf⟧ s
$\mathcal{B}$ ⟦OR(bf_1, bf_2)⟧ s = $b_1 \vee b_2$
$\mathcal{B}$ ⟦AND(bf_1, bf_2)⟧ s = $b_1 \wedge b_2$
$\mathcal{B}$ ⟦IMPL(bf_1, bf_2)⟧ s = $(\neg b_1) \vee b_2$
$\mathcal{B}$ ⟦EQUIV(bf_1, bf_2)⟧ s = $(b_1 \wedge b_2) \vee ((\neg b_1) \wedge (\neg b_2))$
où b_i = $\mathcal{B}$ ⟦bf_i⟧ s

d. Les formules d'état :

Soit la fonction sémantique suivante :

$\mathcal{S} : \mathbf{Sf} \rightarrow \mathbf{S} \rightarrow \mathbf{B}$

qui, à une formule logique associée à un état, renvoie la valeur de vérité (booléenne) de la formule considérée.

$\mathcal{S}[\![\text{Assert(bf)}]\!]\ s = \mathcal{B}[\![\text{bf}]\!]\ s$
$\mathcal{S}[\![\text{Exi(x, sf)}]\!]\ s = \exists\ s'.\ (s'(z) = s(z)\ \forall\ z \in \mathbf{V} \setminus \{x\}) \wedge (\mathcal{S}[\![\text{sf}]\!]\ s')$
$\mathcal{S}[\![\text{Uni(x, sf)}]\!]\ s = a$
où $a = \mathcal{S}[\![\text{sf}]\!]\ s'$
et $\forall\ s'.\ s'(z) = s(z)\ \forall\ z \in \mathbf{V} \setminus \{x\}$
$\mathcal{S}[\![\text{NOT(sf)}]\!]\ s = \neg\ f$
où $f = \mathcal{S}[\![\text{sf}]\!]\ s$
$\mathcal{S}[\![\text{OR}(sf_1, sf_2)]\!]\ s = f_1 \vee f_2$
$\mathcal{S}[\![\text{AND}(sf_1, sf_2)]\!]\ s = f_1 \wedge f_2$
$\mathcal{S}[\![\text{IMPL}(sf_1, sf_2)]\!]\ s = (\neg\ f_1) \vee f_2$
$\mathcal{S}[\![\text{EQUIV}(sf_1, sf_2)]\!]\ s = (f_1 \wedge f_2) \vee ((\neg\ f_1) \wedge (\neg\ f_2))$
où $f_i = \mathcal{S}[\![sf_i]\!]\ s$

4.3.2. Sémantique des formules temporelles

Ici, nous avons décidé, par souci de clarté, de faire la distinction entre les quantificateurs et les opérateurs ; cette seconde partie étant elle-même divisée entre les opérateurs futurs et les opérateurs passés. Nous développerons également les versions strictes des opérateurs, à savoir celles qui n'évaluent pas l'instant courant.

a. Généralités :

Soit la fonction sémantique suivante :

$\mathcal{T} : \mathbf{tf} \rightarrow \mathbf{m} \rightarrow \mathbf{I} \rightarrow \mathbf{B}$

qui, à une formule de logique temporelle associée à un modèle (histoire), évalue, pour un instant donné, la valeur de vérité (booléenne) de cette formule.

$\mathcal{T}[\![\text{TAssert(sf)}]\!]\ m\ j = a$
où $a = \mathcal{S}[\![\text{sf}]\!]\ s$
et $s = m[j]$
(l'état courant à la position/l'instant j du modèle M)

b. Les quantificateurs :

$$\mathcal{T}[\text{Exi}(x, tf)]\, m\ j = a$$
$$\text{où } a = \mathcal{T}[tf]\, m'j$$
$$\text{et } m' = m[j/s']$$
$$\text{et } \exists\, s'.\ s'(z) = s(z)\ \forall\ z \in \mathbf{V} \setminus \{x\}$$
$$\text{et } s = m[j]$$

On choisit donc un état dans lequel une nouvelle variable x est liée, toutes les autres restant identiques, à un renommage près. En effet, si tf contenait des occurrences d'une variable libre identifiées par x, celles-ci doivent être renommées pour éviter toute confusion.

$$\mathcal{T}[\text{Uni}(x, tf)]\, m\ j = a$$
$$\text{où } a = \mathcal{T}[tf]\, m'j$$
$$\text{et } m' = m[j/s']$$
$$\text{et } \forall\ s'.\ s'(z) = s(z)\ \forall\ z \in \mathbf{V} \setminus \{x\}$$
$$\text{et } s = m[j]$$

C'est ici tous les états qui sont modifiés, en intégrant une nouvelle variable liée x, toutes autres restant identiques, à un renommage près.

c. Les opérateurs futurs :

$$\mathcal{T}[\text{TOP}_1(o, tf)]\, m\ j = \mathcal{T}[tf]\, m(j+1)$$
$$\mathcal{T}[\text{TOP}_1(\Box, tf)]\, m\ j = \forall\ k \geq j\ .\ \mathcal{T}[tf]\, mk$$
$$\mathcal{T}[\text{TOP}_1(\Diamond, tf)]\, m\ j = \exists\ k \geq j\ .\ \mathcal{T}[tf]\, mk$$
$$\mathcal{T}[\text{TOP}_2(U, p, q)]\, m\ j = (\exists\ k \geq j\ .\ \mathcal{T}[q]\, mk) \wedge (\forall\ i : j \leq i < k\ .\ \mathcal{T}[p]\, mi)$$
$$\mathcal{T}[\text{TOP}_2(W, p, q)]\, m\ j = (\mathcal{T}[\text{TOP}_1(\Box, p)]\, mj) \vee (\mathcal{T}[\text{TOP}_2(U, p, q)]\, mj)$$

Les versions strictes :

$$\mathcal{T}[\text{TOP}_1(\hat{\sim}, tf)]\, m\ j = \forall\ k > j\ .\ \mathcal{T}[tf]\, mk$$
$$\text{ou } \mathcal{T}[\text{TOP}_1(o, \text{TOP}_1(\Box, tf))]\, mj$$
$$\text{ou } \mathcal{T}[\text{TOP}_1(\Box, tf)]\, m(j+1)$$
$$\mathcal{T}[\text{TOP}_1(\hat{\Diamond}, tf)]\, m\ j = \mathcal{T}[\text{TOP}_1(\Diamond, tf)]\, m(j+1)$$
$$\mathcal{T}[\text{TOP}_2(\hat{U}, p, q)]\, m\ j = \mathcal{T}[\text{TOP}_2(U, p, q)]\, m(j+1)$$
$$\mathcal{T}[\text{TOP}_2(\hat{W}, p, q)]\, m\ j = \mathcal{T}[\text{TOP}_2(W, p, q)]\, m(j+1)$$

d. Les opérateurs passés :

$$\mathcal{T}[\mathrm{TOP}_1(\bullet, tf)]\, m\ j = \mathcal{T}[tf]\, m(j-1) \quad \text{si } j > 0 \text{ faux sinon}$$
$$\mathcal{T}[\mathrm{TOP}_1(\blacksquare, tf)]\, m\ j = \forall\ k : 0 \le k \le j\ .\ \mathcal{T}[tf]\, mk$$
$$\mathcal{T}[\mathrm{TOP}_1(\blacklozenge, tf)]\, m\ j = \exists\ k : 0 \le k \le j .\ \mathcal{T}[tf]\, mk$$
$$\mathcal{T}[\mathrm{TOP}_2(\mathcal{S}, p, q)]\, m\ j = (\exists\ k : 0 \le k \le j .\ \mathcal{T}[q]\, mk) \wedge (\forall\ i : k \le i < j\ .\ \mathcal{T}[p]\, mi)$$
$$\mathcal{T}[\mathrm{TOP}_2(\mathcal{B}, p, q)]\, m\ j = (\mathcal{T}[\mathrm{TOP}_1(\blacksquare, p)]\, mj) \vee (\mathcal{T}[\mathrm{TOP}_2(\mathcal{S}, p, q)]\, mj)$$
$$\mathcal{T}[\mathrm{TOP}_1(\tilde{\bullet}, tf)]\, m\ j = (j=0) \vee (\mathcal{T}[tf]\, m(j-1))$$

Les versions strictes :

$$\mathcal{T}[\mathrm{TOP}_1(\wedge\Box, tf)]\, m\ j = (j=0) \vee (\mathcal{T}[\mathrm{TOP}_1(\blacksquare, tf)]\, m(j-1))$$
$$\mathcal{T}[\mathrm{TOP}_1(\hat{\blacklozenge}, tf)]\, m\ j = \mathcal{T}[\mathrm{TOP}_1(\blacklozenge, tf)]\, m(j-1)$$
$$\mathcal{T}[\mathrm{TOP}_2(\hat{S}, p, q)]\, m\ j = \mathcal{T}[\mathrm{TOP}_2(\mathcal{S}, p, q)]\, m(j-1)$$
$$\mathcal{T}[\mathrm{TOP}_2(\hat{B}, p, q)]\, m\ j = (j=0) \vee (\mathcal{T}[\mathrm{TOP}_2(\mathcal{B}, p, q)]\, m(j-1))$$

4.4. Syntaxe concrète

Voici (enfin) la syntaxe telle qu'on peut concrètement la rencontrer lorsqu'on manipule les formules de logique temporelle décrites dans ce chapitre. Tous les éléments sont des symboles terminaux, à l'exception des symboles propres à la notation BNF ([], {}, *, $^+$).

```
< B > ::=      T | F
< I > ::=      1 | 2 | 3 | …
< Ch > ::=     a | b | c | …
< Var > ::=    < Ch >+
< Varlist > ::= ∅ | < Var > [, < Varlist >]
< V > ::=      {< Varlist >}
< Aff > ::=    < Var > : < I > | < Var > : < B >
< Afflist > ::= ∅ | < Aff > [, < Afflist >]
< S > ::=      < Afflist >
< opu > ::=    - | ¬
< opb > ::=    + | - | ∧ | ∨
< Exp > ::=    < Var > | < opu > < Exp > | < Exp > < opb > < Exp > |
               (< Exp >)
< opp > ::=    < | > | =
< Af > ::=     < Var > | < Exp > < opp > < Exp >
< Boolconnect > ::= ∧ | ∨ | ⇒ | ⇔
< optu > ::=   o | □ | ◊ | • | □ | ♦ | •̃ | ^□ | ◊̂ | ∼̂ | ♦̂
< optb > ::=   U | W | S | B | Û | Ŵ | Ŝ | B̂
```

```
< Bf > ::=     < Af > | (< Bf >) | ¬ < Bf > | < Bf >
               < Boolconnect > < Bf >
< Sf > ::=     < Bf > | ∃ < Var > : < Sf > | ∀ < Var > : < Sf > |
               ¬ < Sf > | (< Sf >) | < Sf > < Boolconnect > < Sf >
< Tf > ::=     < Sf > | ∃ < Var > : < Tf > | ∀ < Var > : < Tf > |
               ¬ < Tf > | (< Tf >) | < Tf > < Boolconnect > < Tf > |
               < optu > < Tf > | < Tf > < optb > < Tf >
```

4.5. Patterns

4.5.1. Formules temporelles

Maintenant que nous avons défini formellement les éléments de logique temporelle qui nous intéressent , il nous semble intéressant de décrire effectivement et opérationnellement la sémantique des patterns utilisés en pratique. Nous utilisons pour cela deux formalismes :

- La sémantique dénotationnelle déjà utilisée plus haut.
- Un langage fonctionnel pour « opérationnaliser » les définitions sémantiques. Cette représentation fonctionnelle nous semble essentielle en ce sens qu'elle permet clairement d'appréhender le comportement des différents patterns, elle respecte la sémantique, et elle est facilement traduisible en RUSSEL. Ces deux derniers aspects seront d'ailleurs traités plus loin.

Rappelons d'abord quels sont ces patterns :

- Achieve : modélise une situation dans laquelle, si une condition P de déclenchement est remplie, alors une condition Q sera vraie dans le futur.
- Cease : modélise une situation dans laquelle, si une condition P de déclenchement est remplie, alors une condition Q cessera d'être vraie dans le futur.
- Maintain : modélise une situation dans laquelle, si une condition P de déclenchement est remplie, alors une condition Q sera toujours vraie dans le futur.
- Avoid : modélise une situation dans laquelle, si une condition P de déclenchement est remplie, alors une condition Q sera toujours fausse dans le futur.
- Next : modélise une situation dans laquelle, si une condition P de déclenchement est remplie, alors une condition Q sera vraie au moment suivant.
- Until : modélise une situation dans laquelle, si une condition P de déclenchement est remplie, alors la condition Q sera toujours vraie jusqu'à la survenance (garantie) d'un événement rendant la condition R vraie.
- Unless : modélise une situation dans laquelle, si une condition P de déclenchement est remplie, alors la condition Q sera toujours vraie jusqu'à la survenance (**non** garantie) d'un événement rendant la condition R vraie. Si R n'est jamais vraie, alors Q est maintenue.

Le cas échéant, chacun de ces patterns peuvent être bornés temporellement. C'est d'ailleurs souvent une « obligation opérationnelle », dans la mesure où, par

exemple pour un « Achieve », il ne sert à rien en pratique de garantir qu'une condition sera remplie dans l'avenir, si le délai nécessaire est (potentiellement) infini.

Les formules correspondantes à ces différents patterns sont donc les suivantes. Notons que, en logique temporelle, il existe une implication « faible » ($\rightarrow$) et une « forte » ($\Rightarrow$). Néanmoins, les expressions ($\Box$ (P $\rightarrow$ Q)) et (P $\Rightarrow$ Q) sont sémantiquement équivalentes.

pour Achieve :

$$\Box\ (P \rightarrow \Diamond_{\leq t}\ Q)$$

pour Cease :

$$\Box\ (P \rightarrow \Diamond_{\leq t}\ \neg\ Q)$$

pour Maintain :

$$\Box\ (P \rightarrow \Box_{\leq t}\ Q)$$

pour Avoid :

$$\Box\ (P \rightarrow \Box_{\leq t}\ \neg\ Q)$$

pour Next :

$$\Box\ (P \rightarrow o\ Q)$$

pour Until :

$$\Box\ (P \rightarrow Q\ U_{\leq t}\ R)$$

pour Unless :

$$\Box\ (P \rightarrow Q\ W_{\leq t}\ R)$$

C'est à chaque fois la partie à droite de l'implication qui nous intéresse, et que nous devrons « opérationnaliser ». C'est en effet cette partie qui comprend la formule de logique temporelle proprement dite. La partie de gauche (le « P ») n'est qu'une condition de déclenchement. Nous verrons dans le chapitre suivant comment les choix d'architecture que nous avons fait imposent de traiter séparément les parties de gauche et de droite. Pour l'instant, retenons seulement que nous devons donner l'équivalent des « Eventually, Always, Next, Until et Unless », et éventuellement de leur négation. Cet équivalent doit être, à terme, en RUSSEL, et doit bien entendu respecter scrupuleusement la sémantique originale.

Pour cela, nous nous proposons d'utiliser le langage fonctionnel CaML. Nous verrons par la suite que ce choix permet d'être clair et concis, aussi bien dans les preuves de correction que dans la traduction des fonctions CaML en RUSSEL.

4.5.2. Fonctions temporelles – CaML

a. Les Notations :

- q, r : Ce sont les équivalents, en CaML, des formules Q et R reprises dans les patterns. Rappelons que ce sont des formules de logique classique, conjonctions, disjonctions, implications et équivalences… Pour une raison de concision, nous avons décidé que q et r seraient traitées comme des fonctions, qui reçoivent une valuation (liste des variables reprises dans q et r et leurs valeurs booléennes correspondantes) et renvoient la valeur de la formule (booléenne, évidemment).

- m : Le modèle, ou histoire. Il représente en fait l'évolution du système, en reprenant la liste des couples (événement, valeur).

b. Les types CaML :

- Valuation : Une valuation est une liste des valeurs de vérité de chaque événement considérées à un instant spécifique. Cette liste correspond donc à une « tranche de temps » et reprend, pour cette « tranche de temps », tous les évènements considérés et leur valeur booléenne. Ce qui correspond au type CaML suivant :

```
type val = (string * bool) list ;;
```

- Modèle : Un modèle (ou histoire) du système est une liste de valuations, représentant l'évolution du système par le biais de l'évolution des valeurs de vérité des éléments du système.

```
type mod = val list ;;
```

c. Les fonctions CaML :

- **Achieve :** $\Box$ **(P** $\rightarrow$ $\Diamond_{\leq t}$ **Q)**

La sémantique associée au pattern est :

$$\mathcal{T}[\![TOP_1(\Diamond, Q)]\!]\, mj = \exists\ k : j \leq k \leq j+t\ .\ \mathcal{T}[\![Q]\!]\, mk$$

Et la fonction CaML :

```
Fct : Achieve qm
Type : (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Achieve q = function [e] → qe
                       | e :: l → qe || Achieve ql ;;
```

Preuve de correction :

a. Relation bien fondée sur la structure des termes : le paramètre d'induction est le modèle m, qui est une liste. Nous raisonnons donc par induction sur la structure de liste.

b. Preuve :

- base : m = [e] ⇒ ◊ Q = qe
- induction : m = e :: l ⇒ ◊ Q = qe || Achieve ql

```
   ⇒    qe = true   ⇒ (qe || Achieve ql) = true
                    ⇒ ◊ Q = true
        qe = false  ⇒ (qe || Achieve ql) = Achieve ql
                    ⇒ ◊ Q = Achieve ql
```

Intuitivement, on dira que si le modèle ne contient qu'une valuation, alors la valeur de vérité de Achieve est celle de Q appliqué à cette valuation. Si le modèle contient plusieurs valuations, Achieve sera vrai si la valeur de vérité de Q appliqué à la valuation courante est vraie, ou si elle est vraie pour (au moins) une des valuations suivantes.

- **Cease :** $\Box\ (P \rightarrow \Diamond_{\leq t}\ \neg\ Q)$

La sémantique associée au pattern est :

$$\mathcal{T}[\![TOP_1(\Diamond, \neg Q)]\!]\ mj = \exists\ k : j \leq k \leq j+t\ .\ \mathcal{T}[\![\neg Q]\!]\ mk$$

Et la fonction CaML :

```
Fct : Cease qm
Type : (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Cease q = function [e] → ¬(qe)
                     | e :: l → ¬(qe) || Cease ql ;;
```

Preuve de correction :

a. Relation bien fondée sur la structure des termes : le paramètre d'induction est le modèle m, qui est une liste. Nous raisonnons donc par induction sur la structure de liste.

b. Preuve :

- base : m = [e] ⇒ ◊ ¬Q = ¬qe
- induction : m = e :: l ⇒ ◊ ¬Q = ¬qe || Cease ql

```
   ⇒    qe = false  ⇒ (¬qe || Cease ql) = true
                    ⇒ ◊ ¬Q = true
        qe = true   ⇒ (¬qe || Cease ql) = Cease ql
                    ⇒ ◊ ¬Q = Cease ql
```

Intuitivement, on dira que si le modèle ne contient qu'une valuation, alors la valeur de vérité de Cease est celle de ¬Q appliqué à cette valuation. Si le modèle contient plusieurs valuations, Cease sera vrai si la valeur de vérité de Q appliqué à la valuation courante est fausse, ou si elle est fausse pour (au moins) une des valuations suivantes.

- **Maintain : □ (P → □≤ t Q)**

La sémantique associée au pattern est :

$$\mathcal{T}[\![TOP_1(\Box, Q)]\!]\, mj = \forall\ k : j \le k \le j+t\ .\ \mathcal{T}[\![Q]\!]\, mk$$

Et la fonction CaML :

```
Fct : Maintain qm
Type : (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Maintain q = function [e] → qe
                        | e :: l → qe && Maintain ql ;;
```

Preuve de correction :

a. Relation bien fondée sur la structure des termes : le paramètre d'induction est le modèle m, qui est une liste. Nous raisonnons donc par induction sur la structure de liste.

b. Preuve :

- base : m = [e] ⇒ □ Q = qe
- induction : m = e :: l ⇒ □ Q = qe && Maintain ql

```
   ⇒     qe = false    ⇒ (qe && Maintain ql) = false
                       ⇒ □ Q = false

         qe = true ⇒ (qe && Maintain ql) = Maintain ql
                   ⇒ □ Q = Maintain ql
```

Intuitivement, on dira que si le modèle ne contient qu'une valuation, alors la valeur de vérité de Maintain est celle de Q appliqué à cette valuation. Si le modèle contient plusieurs valuations, Maintain sera vrai si la valeur de vérité de Q appliqué à la valuation courante est vraie, et si elle est vraie pour toutes les valuations suivantes.

- **Avoid : □ (P → □≤ t ¬ Q)**

La sémantique associée au pattern est :

$$\mathcal{T}[\![TOP_1(\Box, \neg Q)]\!]\, mj = \forall\ k : j \le k \le j+t\ .\ \mathcal{T}[\![\neg Q]\!]\, mk$$

Et la fonction CaML :

```
Fct : Avoid qm
Type : (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Avoid q = function [e] → ¬qe
                    | e :: l → ¬qe && Avoid ql ;;
```

Preuve de correction :

a. Relation bien fondée sur la structure des termes : le paramètre d'induction est le modèle m, qui est une liste. Nous raisonnons donc par induction sur la structure de liste.

b. Preuve :

- base : m = [e] ⇒ □ ¬Q = ¬qe
- induction : m = e :: l ⇒ □ ¬Q = ¬qe && Avoid ql
  ⇒ qe = true ⇒ (¬qe && Avoid ql) = false
  ⇒ □ ¬Q = false
  qe = false ⇒ (¬qe && Avoid ql) = Avoid ql
  ⇒ □ ¬Q = Avoid ql

Intuitivement, on dira que si le modèle ne contient qu'une valuation, alors la valeur de vérité de Avoid est celle de ¬Q appliqué à cette valuation. Si le modèle contient plusieurs valuations, Avoid sera vrai si la valeur de vérité de Q appliqué à la valuation courante est fausse, et si elle est fausse pour toutes les valuations suivantes.

- **Until :** **□ (P → Q $\mathcal{U}_{\leq t}$ R)**

La sémantique associée au pattern est :

$$\mathcal{T}[\![TOP_2(\mathcal{U}, Q, R)]\!]\, mj = (\exists\ k : j \leq k \leq j+t\ .\ \mathcal{T}[\![R]\!]\, mk) \wedge (\forall\ i : j \leq i < k\ .\ \mathcal{T}[\![Q]\!]\, mi)$$

Et la fonction CaML :

```
Fct : Until qrm
Type : (val → bool) → (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Until qr= function [e] → re
            | e :: l → if (re) then true
                         else (qe) && (Until qrl) ;;
```

Preuve de correction :

a. Relation bien fondée sur la structure des termes : le paramètre d'induction est le modèle m, qui est une liste. Nous raisonnons donc par induction sur la structure de liste.

b. Preuve :

- base : m = [e] ⇒ Q 𝒰 R = re
- induction : m = e :: l ⇒ Q 𝒰 R = if (re) then true else (qe) && (Until qrl)

  ⇒ re = true ⇒ if (re) then true else (qe) && (Until qrl) = true
  ⇒ Q 𝒰 R = true

  re = false ⇒ if (re) then true else (qe) && (Until qrl) = (qe) && (Until qrl)

  ⇒ qe = true
  ⇒ (qe) && (Until qrl) = Until qrl
  ⇒ Q 𝒰 R = Until qrl

  qe = false
  ⇒ (qe) && (Until qrl) = false
  ⇒ Q 𝒰 R = false

Intuitivement, on dira que si le modèle ne contient qu'une valuation, alors la valeur de vérité de Until est celle de R appliqué à cette valuation. Si le modèle contient plusieurs valuations, Until sera vrai si Q est maintenu vraie tant que R ne l'est pas. On remarquera au passage que cette formulation fait ressortir le cas qui invalide le pattern, à savoir celui où Q et R sont faux en même temps.

- **Unless : □ (P → Q 𝒲≤ t R)**

La sémantique associée au pattern est :

$$\mathcal{T}[\![TOP_2(\mathcal{W}, Q, R)]\!]\, mj = (\mathcal{T}[\![TOP_1(\square, Q)]\!]\, mj) \vee (\mathcal{T}[\![TOP_2(\mathcal{U}, Q, R)]\!]\, mj)$$

Et la fonction CaML :

```
Fct : Unless qrm
Type : (val → bool) → (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Unless qr= function [e] → re || qe
            | e :: l → if (re) then true
                          else (qe) && (Unless qrl) ;;
```

Preuve de correction :

a. Relation bien fondée sur la structure des termes : le paramètre d'induction est le modèle m, qui est une liste. Nous raisonnons donc par induction sur la structure de liste.

b. Preuve :

- base : m = [e] ⇒ Q W R = re || qe
- induction : m = e :: l ⇒ Q W R = if (re) then true else (qe) && (Unless qrl)

  ⇒ re = true ⇒ if (re) then true else (qe) && (Unless qrl) = true
  ⇒ Q W R = true

  re = false ⇒ if (re) then true else (qe) && (Unless qrl) = (qe) && (Unless qrl)

  ⇒ qe = true
  ⇒ (qe) && (Unless qrl) = Unless qrl
  ⇒ Q W R = Unless qrl

  qe = false
  ⇒ (qe) && (Unless qrl) = false
  ⇒ Q W R = false

Intuitivement, le raisonnement est identique à celui développé pour Until, à ceci près que R peut ne jamais être vrai, auquel cas Q est maintenu jusqu'au terme du modèle.

- **Next :** □ **(P → o Q)**

La sémantique associée au pattern est :

$\mathcal{T}[\![TOP_1(o, Q)]\!]\, mj = \mathcal{T}[\![Q]\!]\, m(j+1)$

Et la fonction CaML :

```
Fct : Next qm
Type : (val → bool) → mod → bool
cond : ∅

let next q = function e :: l :: tail → ql
                | _ → false ;;
```

Ce qui correspond bien au cas où l'on évalue Q pour l'instant suivant (l'élément suivant du modèle). Cette fonction n'est pas récursive, la preuve par induction n'a donc pas lieu d'être.

5. ORKA

5.1. Introduction – Obstacle Recognition with Kaos based on Asax

C'est dans ce chapitre que nous traiterons des dernières phases du processus de traduction qui, rappelons-le, comporte 4 phases. La définition des différents patterns et leur sémantique a d'ores et déjà été définie au chapitre précédent. Nous avons également proposé une version CaML de ces différents patterns, et démontré que ces derniers étaient sémantiquement équivalents aux fonctions sémantiques.

L'usage du CaML permet de travailler sur une représentation « intermédiaire », plus proche d'une version « opérationnelle », plus facile à manipuler, et simple en ce qui concerne les preuves de correction.

Cependant, ces fonctions CaML ne peuvent être reprises telles qu'elles sont présentées au chapitre précédent. Outre le fait que RUSSEL ne soit pas un langage fonctionnel mais un langage « orienté règles », la grande différence réside dans les représentations des « histoires » ou « traces d'exécution » des programmes à vérifier. En effet, en CaML, nous travaillons sur des « listes de valuations », et, en RUSSEL, sur des traces reprenant, en chacun des éléments de ces traces, un événement, une valeur, un timestamp. Nous verrons comment nous pouvons concilier ces points de vue.

Nous attirons également l'attention du lecteur sur un point bien précis : il ne s'agit nullement ici de modéliser/spécifier le comportement attendu du programme (il s'agit alors du travail de l'utilisateur de KAOS) mais bien de comprendre la sémantique de cette modélisation et de traduire celle-ci en RUSSEL en conservant sa sémantique.

Nous présenterons donc une version « diagrammes » des règles, pour régler une fois pour toute les problèmes purement opérationnels (structure du programme RUSSEL, formats de traces et histoires…) sans nous préoccuper encore de la syntaxe proprement dite. Enfin, nous présenterons la traduction systématique, en RUSSEL, des différents éléments utilisés depuis le début du processus de traduction.

5.2. Notions d'événement

Par événement, nous désignons n'importe quel changement d'état du système, représenté par un couple (nom, valeur), où « nom » dénote un « composant » du système, comme un attribut ou une association d'objets, et « valeur » désigne soit une valeur d'un attribut, soit la notion d'existence d'une association. Dans la trace d'exécution du système, nous ne trouverons donc que des couples (nom, valeur) et tout événement du système doit être réduit à cette expression (nom, valeur).

On peut donc dire que tout attribut du système qui change de valeur correspond à un événement. Toute « création » d'un nouvel attribut correspond à la

modification d'une valeur (de type « undef. »), toute « suppression » correspond à la modification de la valeur actuelle en « undef. ».

Le problème, cependant, concerne les relations entre les objets du modèle Kaos. Comment exprimer la création/modification/suppression d'une association entre objets uniquement à l'aide d'attributs ?

Nous pouvons à ce sujet faire l'analogie avec les concepts de bases de données relationnelles et schémas entités-relations-attributs.

Dans le cas d'entités ou d'objets pouvant être exprimés sous forme d'attribut :

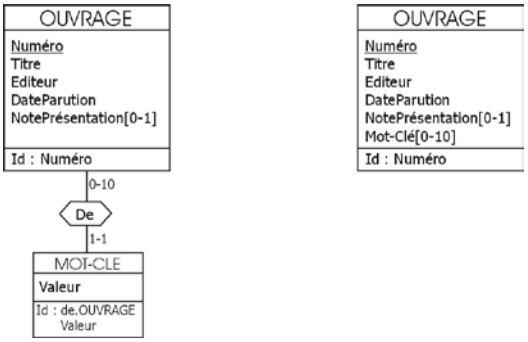


fig. 5.1 – transformation d'entité en attribut

Le « concept » de mot-clé est ici représenté sous la forme d'un attribut. L'association Ouvrage/Mot-Clé est supprimée.

Dans le cas d'associations pouvant être exprimées sous forme d'attributs :

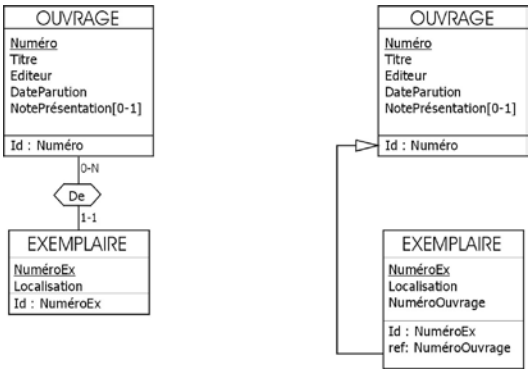


fig. 5.2 – transformation d'association en attribut

L'association Ouvrage/Exemplaire est ici remplacée avantageusement par l'attribut (la référence) NuméroOuvrage.

L’avantage de ce type de transformation est bien entendu son caractère systématique. On peut définir différents cas d’utilisation, principalement basés sur les cardinalités, permettant de ramener une association à des attributs. Voici ces cas d’utilisation :

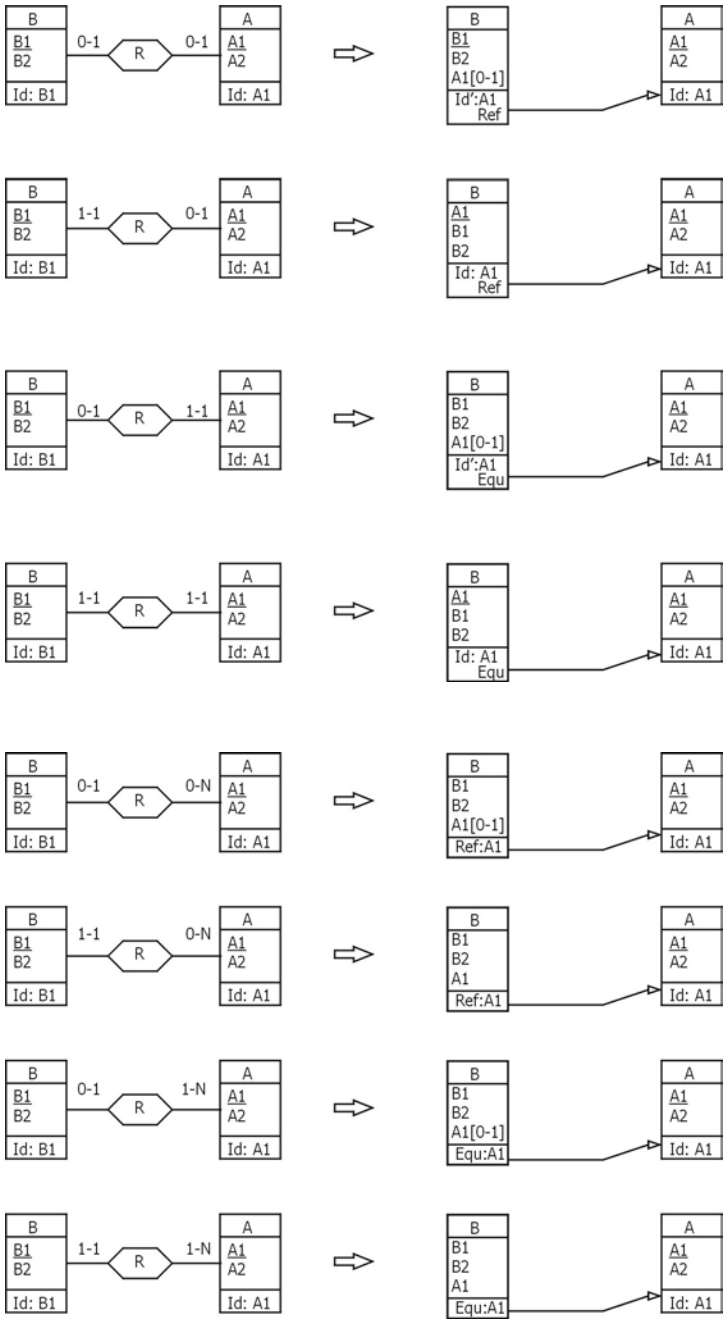


fig. 5.3 – patterns de transformation

La liste n’est pas exhaustive, mais elle donne un bon aperçu des possibilités.

5.3. Méthodologie – Design d'implémentation

Deux « lignes directrices » on été suivie lors de l'élaboration du traducteur : « l'approche Patterns » et la règle « Supervisor ». Détaillons ces deux concepts :

5.3.1. Les patterns, les fonctions CaML

L'approche « patterns » a déjà été développée longuement dans les chapitres précédents. Reprenons donc ces patterns :

Achieve :	□ (P → $\Diamond_{\leq t}$ Q)
Cease :	□ (P → $\Diamond_{\leq t}$ ¬ Q)
Maintain :	□ (P → $\Box_{\leq t}$ Q)
Avoid :	□ (P → $\Box_{\leq t}$ ¬ Q)
Until :	□ (P → Q $U_{\leq t}$ R)
Unless :	□ (P → Q $W_{\leq t}$ R)
Next :	□ (P → o Q)

L'idée directrice est ici de distinguer deux parties dans ces patterns : la P-Part et la Q-Part.

a. La P-Part :

La « partie P » ou « Condition P » représente en réalité une condition de déclenchement d'une règle. C'est une conjonction d'évènements ou un événement simple. Une disjonction d'évènements est possible, mais nous lui préférons une représentation via deux (ou plusieurs) conditions distinctes. A ce stade de développement, les assertions P sont donc des formules de logique classique. En RUSSEL, elles correspondront à des conditions de déclenchement des règles. Toutes ces conditions seront regroupées dans une règle Supervisor.

b. La Q-Part :

La « partie Q » représente la règle en tant que tel. C'est la formule de logique temporelle « à droite », qui correspondra effectivement à une règle en RUSSEL.
Cette « partie Q » est elle-même décomposable en ses éléments de base, à savoir un connecteur de logique temporelle, et une ou deux formules, qui sont, à ce stade, des formules de logique classique. Ces formules sont de 5 « types » :

- L'événement simple : e
- La conjonction : $e_1 \wedge e_2 \wedge \ldots \wedge e_n$
- La disjonction : $e_1 \vee e_2 \vee \ldots \vee e_n$
- L'implication : $e_1 \Rightarrow e_2$, défini comme : $\neg e_1 \vee e_2$
- L'équivalence : $e_1 \Leftrightarrow e_2$: $(e_1 \wedge e_2) \vee (\neg e_1 \wedge \neg e_2)$

Nous pouvons combiner ces « types » selon les règles suivantes : des conjonctions dans des disjonctions, et des disjonctions dans des implications ou des équivalences.

A chaque « partie Q » correspond une fonction CaML, telle que définie au chapitre précédent. C'est cette fonction qui sera traduite en diagramme plus loin dans ce chapitre, pour être ensuite traduite en RUSSEL.

5.3.2. L'approche Supervisor

Cette approche est la conséquence logique de l'utilisation de RUSSEL. Les programmes écrits en RUSSEL ont en fait un mode de fonctionnement de type « si *condition* alors *action* ». L'idée est ici de définir une règle « de base », reprenant l'ensemble des conditions (P-Part) définies sur le système. La responsabilité de cette règle Supervisor est double : d'une part, elle devra évaluer les différentes conditions, et déclencher les règles correspondantes (Q-Part), d'autre part, elle devra gérer la mémoire du système.

Par mémoire du système, nous désignons en réalité tous les évènements à contrôler, la gestion du « timestamp », la gestion des règles actives…

Parallèlement à la règle Supervisor, nous trouverons un ensemble de règles « RuleX » implémentant les différentes Q-Part's. A chacune de ces règles RuleX correspond également une variable binaire activeRuleX. Cette variable-témoin est nécessaire pour éviter de déclencher une même règle à répétition suite au maintient, a priori normal, de sa condition de déclenchement.

Concrètement, à chacune des règles nécessitant une traduction en RUSSEL, nous aurons, dans Supervisor, une « macro-instruction » de la forme :

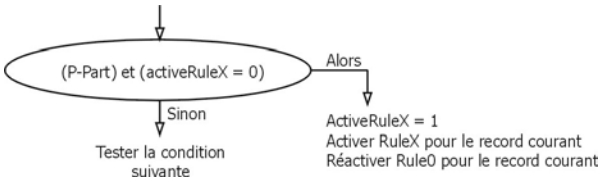


fig. 5.4 – diagramme « p-part »

Concernant l' « initialisation » de Supervisor, rappelons qu'il est nécessaire, vu le format de la trace, de décomposer celle-ci en tranches de temps. L'initialisation correspondant à une tranche de temps définie (et donc à un même timestamp), est donc une mise à jour de toutes les valeurs des évènements considérés dans le système. Schématiquement, ceci correspond à :

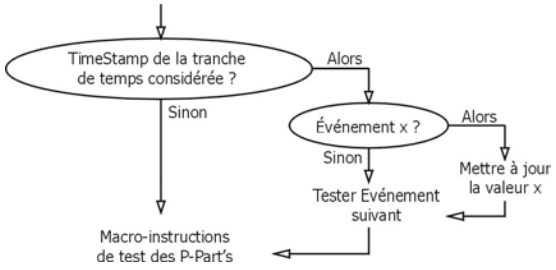


fig. 5.5 – diagramme « extraction des événements »

Intuitivement, ceci peut être expliquer de la sorte : Si nous trouvons toujours le même timestamp, nous sommes donc toujours dans la même tranche de temps.

Comme nous devons évaluer tous les évènements nécessaires à l'exécution des règles RuleX à chaque tranche de temps, on passe en revue ces éléments et on traite ensuite le record suivant.
Cette procédure sera traitée plus en profondeur ci-après, quand il s'agira du concept de « valuation ».

Enfin, la « clôture » de la règle Supervisor, correspondant à la situation où l'on se trouve au début d'une tranche de temps, et que toutes les macro-instructions de test ont été réalisées, se présente sous la forme d'une réactivation de Supervisor, pour un nouveau timestamp, celui de la nouvelle tranche de temps, devenue courante. Les autres paramètres sont les évènements considérés (inchangés) et leurs nouvelles valeurs correspondantes.

On peut dès lors définir l'architecture complète de la règle Supervisor :

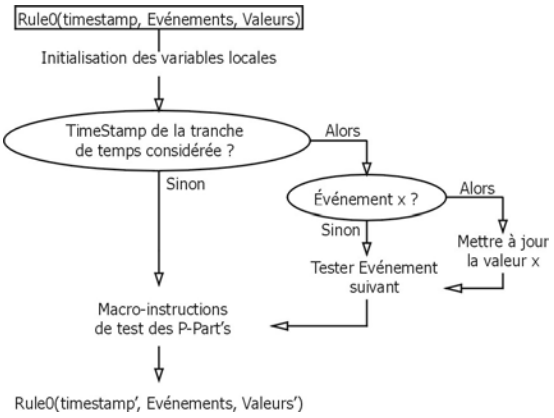


fig 5.6 – diagramme « Supervisor »

Nous avons donc bien, comme prévu, l'évaluation continue des différentes conditions considérées dans le système et le déclenchement des règles correspondantes, par les macro-instructions de test, et la gestion de la mémoire du système, via la mise à jour des valeurs des différents événements. Notons que chacune des règles RuleX recevra en guise de paramètres un sous-ensemble de cette mémoire. Nous verrons en détail l'architecture de ces RuleX's.

5.4. 3ème étape de traduction : les Diagrammes

5.4.1. Raison d'être, vue opérationnelle

Comme nous l'avons déjà mainte fois évoqué, nous suivons un processus de traduction par étapes permettant de se rapprocher de plus en plus d'une vue « opérationnelle ». A chaque étape, donc, nous détaillons de plus en plus le fonctionnement concret des patterns, et nous prenons de plus en plus en considération les concepts purement liés à la technique, le format des objets, les modes de déclenchement…

Les diagrammes, sans pour autant reprendre la syntaxe du RUSSEL, représentent donc le mode de fonctionnement réel des règles RUSSEL, dérivées des patterns de logique temporelle, eux-mêmes exprimés sous la forme de fonctions CaML. Mais lorsque la logique temporelle utilise des « modèles » ou des « histoires », et

que les fonctions CaML travaillent sur des « listes de valuations », le programme écrit en RUSSEL, lui, n'aura comme matière de travail qu'une trace d'exécution d'un programme.

L'équivalence entre cette trace et la liste de valuations utilisée en CaML est cependant assez aisée à obtenir. Chaque élément de la trace comprend un timestamp, un nom et une valeur. On peut donc découper cette trace en tranches de temps (timestamps identiques). Nous avons donc, pour chaque tranche de temps, un ensemble de nom et un ensemble de valeurs correspondantes, ce qui n'est rien d'autre qu'une valuation.

Mais CaML travaille non pas sur des valuations mais sur des listes, le plus souvent finies, de valuations. C'est ici qu'intervient la notion de délai associé aux règles RUSSEL. La trace étant potentiellement infinie (tant que le programme s'exécute), il ne faudra donc prendre en compte que les événements générés endéans un certain délai. Passé ce délai, on n'extrait plus les valuations de la trace, et la liste des valuations (CaML) est considérée comme vide.

5.4.2. Extraction de valuation

Prenons le cas où nous avons deux événements considérés dans une règle. Un nombre plus élevé d'événements ne fait qu'augmenter le nombre de tests réalisés mais ne modifie pas la structure du programme. Schématiquement, nous pouvons définir l'extraction de valuation comme suit :

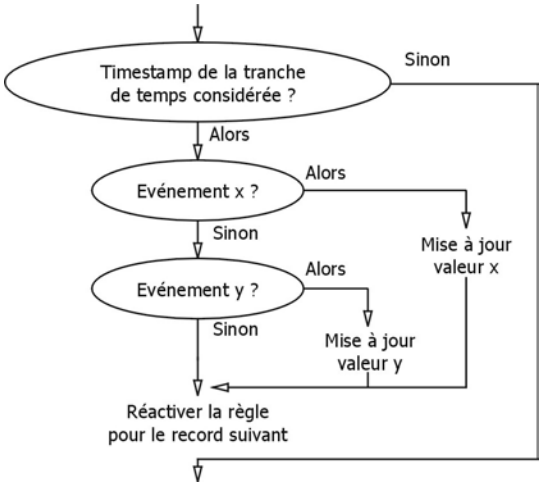


fig. 5.7 – diagramme « extraction de valuation »

Détaillons ce diagramme :

Le test de départ nous assure que nous sommes toujours dans la même tranche de temps. Si ce n'est pas le cas, on quitte la routine d'extraction. Si par contre nous sommes bien dans cette tranche de temps, on examine le nom de l'événement contenu dans le record courant. S'il correspond à l'un des deux (n) événements considérés, alors, l'un des deux (n) tests seront vrais et la valeur de l'événement sera sauvegardée dans la mémoire du système. Si aucun événement considéré ne correspond, on a affaire à un record contenant du « bruit » (un événement inutile à

ce niveau), et on ne mémorise rien. On relance ensuite la routine pour le record suivant.

A la fin de la tranche de temps, la mémoire de la règle (collection de variables locales) contient l'ensemble des événements mis à jour, c'est-à-dire une valuation. On peut donc appliquer la règle définie en CaML sur cette valuation.

5.4.3. Cas général d'utilisation et éléments de base

Nous allons ci-dessous présenter les différents diagrammes correspondant aux patterns utilisés. Pour plus de clarté, nous utiliserons la « convention » suivante : nous considérons d'une part que la condition incluse dans la formule de logique temporelle est une condition « complexe » et qu'elle est donc traduite sous forme de deux conditions simples (Condition1 $\vee$ Condition2). De plus, nous considérons, comme nous l'avons fait lors de l'extraction de valuation, que ces conditions s'expriment relativement à deux événements.

Concernant les éléments de base, nous avons d'ores et déjà à notre disposition l'extraction de valuation. Nous pouvons également introduire d'autres concepts :

a. Le test d'une condition :

Correspond en CaML à *q(e)* où e est la valuation et q la condition. Sans surprise, nous pouvons représenter Test qe comme :

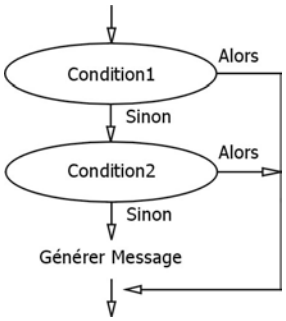


fig. 5.8 – diagramme « test de condition »

Ce qui s'explique aisément : Si la sous condition Condition1 est vraie ou si la sous condition Condition2 est vraie, alors q(e) est vérifiée et l'évaluation se termine. Si les deux sous conditions sont fausses, générer un message d'alarme.

Ce test possède plusieurs variantes. Le Test not(qe) :

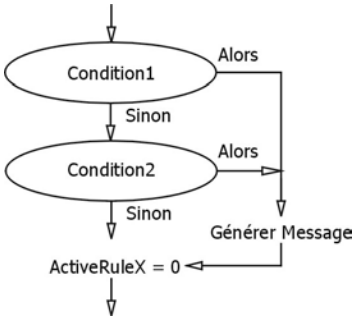


fig. 5.9 – diagramme « test de condition »

Où l'on « permute les sorties ». Le message d'alarme est lancé si l'une des conditions est vraie.

Les cas « itératifs », où l'on remplace l'une des deux sorties (de succès ou d'échec) par « ce qu'il reste à faire ». Ces cas sont détaillés dans les différents patterns.

b. Le test de délai :

Dans la plupart des fonctions CaML, nous trouvons un pattern matching réalisé sur la structure de la liste de valuation. Ce matching correspond, comme nous l'avons déjà évoqué plus haut, à un test sur la variable *delay* de la règle. Ce test est on ne peut plus classique :

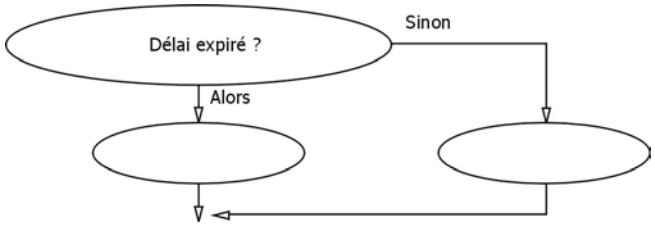


fig. 5.10 – diagramme « test de délai »

Et se place juste à la suite de l'extraction de valuation.

Nous avons donc des diagrammes, correspondant aux patterns, dont la structure est toujours la même. D'abord l'extraction de valuation, ensuite le test de délai, avec l'alternative « positive » (délai non expiré) et l'alternative « négative » (délai expiré). Au sein de ces alternatives, l'« unité de base » est le « Test qe », qui est modifié/recombiné en fonction des spécificités des patterns.

5.4.4. Les patterns

Examinons dès lors ces différents patterns. Notre point de départ est fort logiquement la fonction CaML de chacun d'entre eux.

a. Achieve :

Comme point de départ, nous reprenons la définition CaML :

```
Fct : Achieve qm
Type : (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Achieve q = function [e] → qe
                  | e :: l → qe || Achieve ql ;;
```

Nous avons donc :

- implicitement, l'extraction de la valuation à partir de la trace
- si le délai est épuisé, le test « qe », sinon, le test « qe || Achieve ql »

Ce qui, graphiquement, est représenté par :

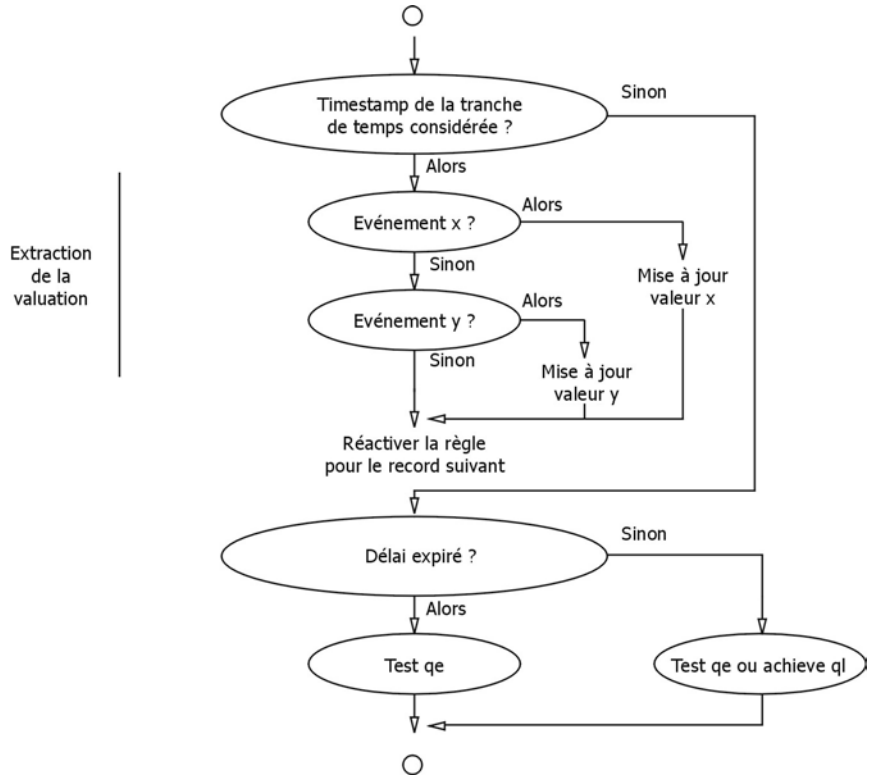


fig. 5.11 – diagramme « Achieve »

Imaginons dès lors que la partie Q reprend une condition élaborée de type Condition1 ∨ Condition2. Le Test qe est représenté par :

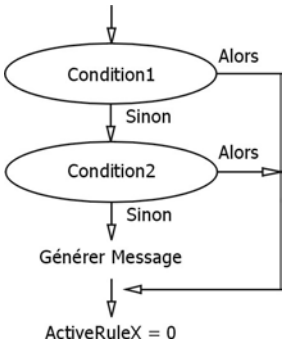


fig. 5.12 – diagramme « test de condition – q(e) »

Et le Test qe || Achieve ql :

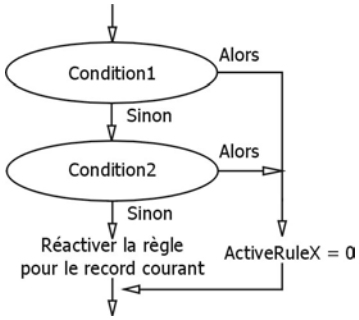


fig. 5.13 – diagramme « test de condition – qe || Achieve ql »

Les autres patterns sont traités de façon similaire.

b. Cease :

```
Fct : Cease qm
Type : (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Cease q = function [e] → ¬(qe)
                  | e :: l → ¬(qe) || Cease ql ;;
```

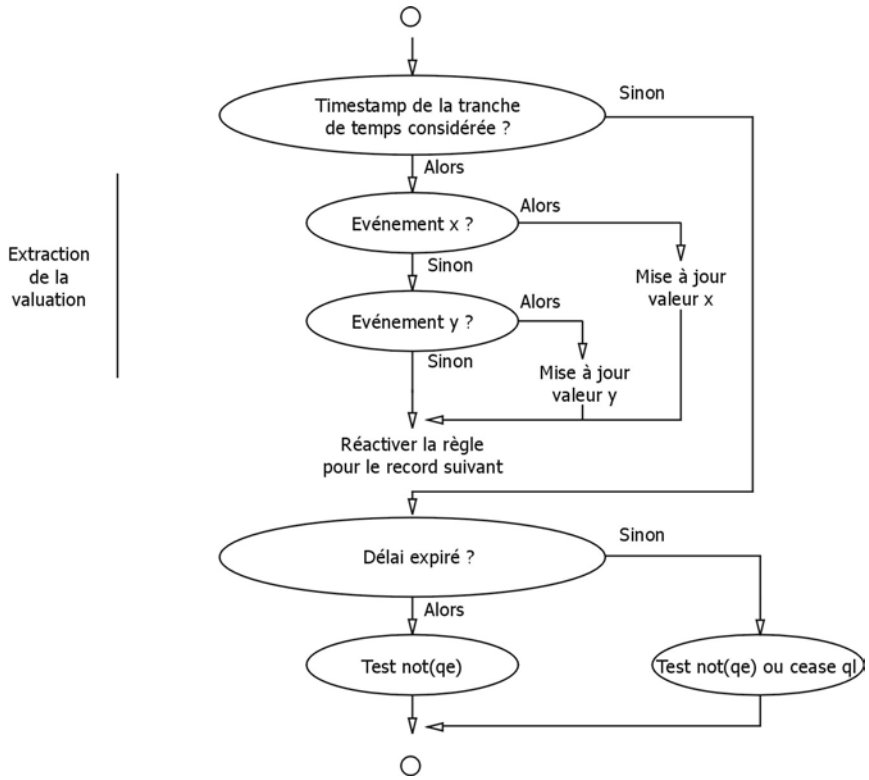


fig. 5.14 – diagramme « Cease »

Pour Test not(qe) :

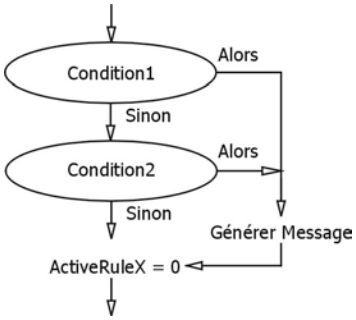


fig. 5.15 – diagramme « test de condition – not q(e) »

Et pour Test not(qe) || Cease ql :

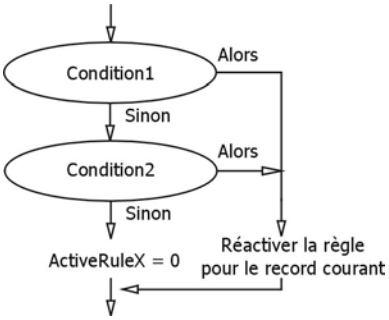


fig. 5.16 – diagramme « test de condition – not qe || Cease ql »

c. Maintain :

```
Fct : Maintain qm
Type : (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Maintain q = function [e] → qe
                   | e :: l → qe && Maintain ql ;;
```

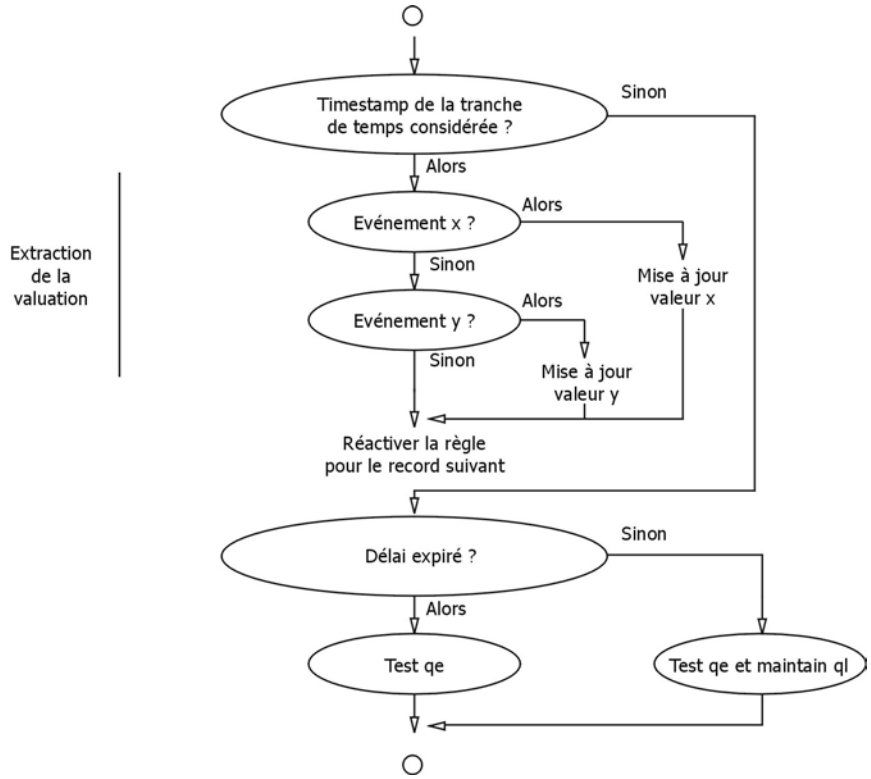


fig. 5.17 – diagramme « Maintain»

La macro-instruction « Test qe » est identique à celle déjà développée plus haut. En ce qui concerne « Test qe et maintain ql », nous avons :

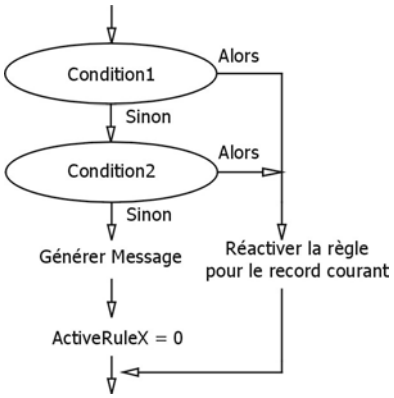


fig. 5.18 – diagramme « test de condition – qe || Maintain ql »

d. Avoid :

```
Fct : Avoid qm
Type : (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Avoid q = function [e] → ¬qe
                  | e :: l → ¬qe && Avoid ql ;;
```

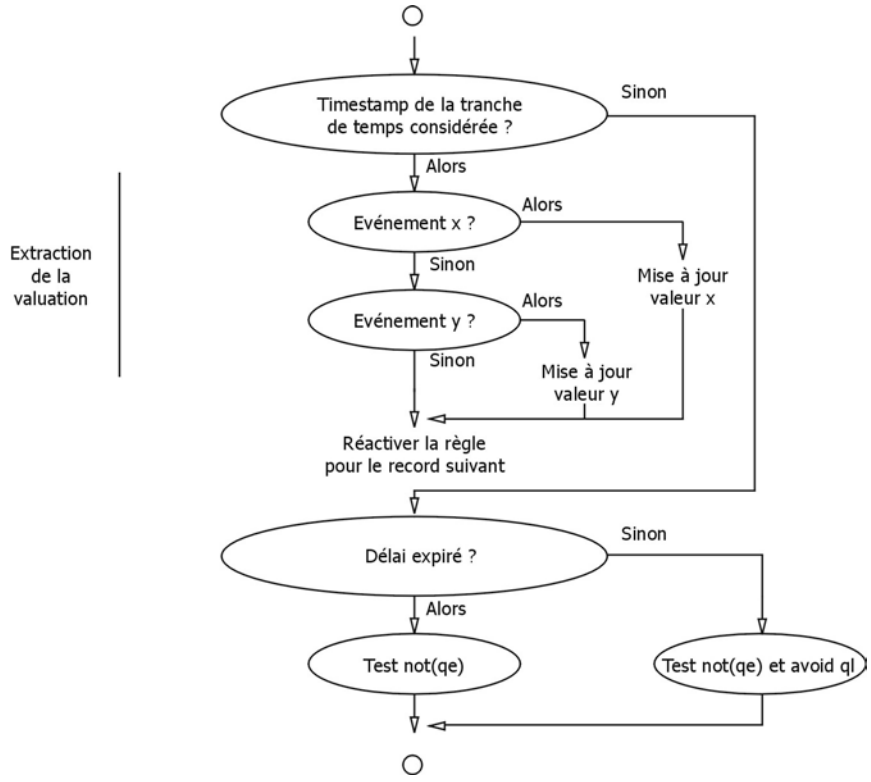


fig. 5.19 – diagramme « Avoid»

A nouveau le « Test not(qe) » a déjà été évoqué. Concernant le « Test not(qe) et avoid ql », cela donne :

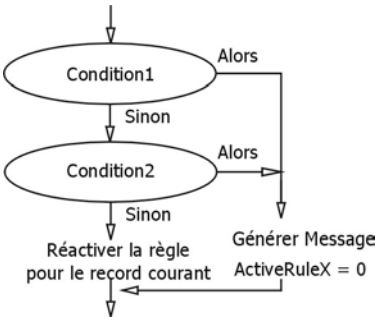


fig. 5.20 – diagramme « test de condition – not qe || Avoid ql »

e. Next :

```
Fct : Next qm
Type : (val → bool) → mod → bool
cond : ∅

let next q = function e :: l :: tail → ql
                | _ → false ;;
```

Le traitement du Next est un peu particulier, en ce sens que la notion de délai est ici a priori superflue. On considère donc ici l'extraction de deux valuations successive, à savoir la valuation courante et la suivante. La macro-instruction « Test ql » est alors immédiate, et est similaire à « Test qe » que nous avons déjà développé.

Le diagramme est donc le suivant :

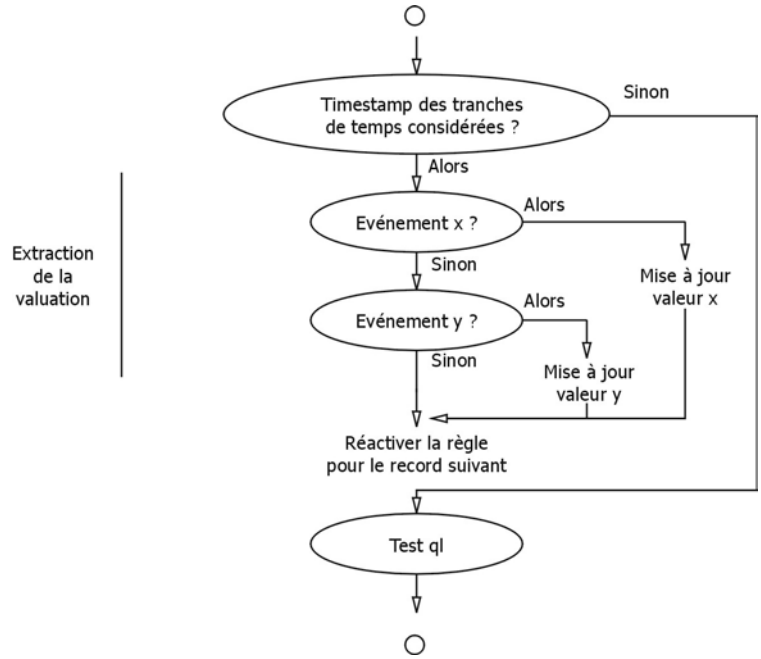


fig. 5.21 – diagramme « Next»

f. Until :

```
Fct : Until qrm
Type : (val → bool) → (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Until qr= function [e] → re
                | e :: l → if (re) then true
                                   else (qe) && (Until qrl) ;;
```

Ici également, les deux patterns Until et Unless bénéficient d'un traitement un peu différent. Si le diagramme général nous est familier, il y a en réalité un niveau de plus dans les macro-instructions, ce que nous désignons ci-après par « Test if ».

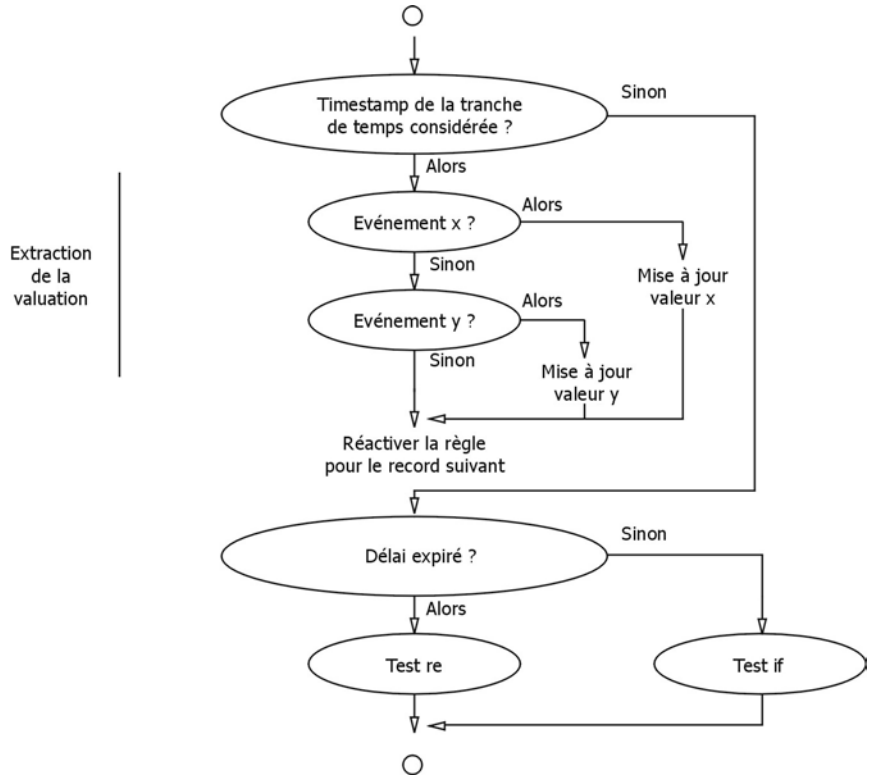


fig. 5.22 – diagramme « Until»

Le « Test re » est équivalent au « Test qe », à un renommage près. Le « Test if » nécessite un développement supplémentaire :

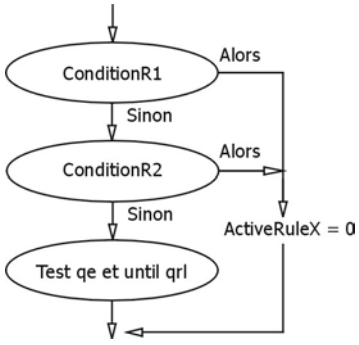


fig. 5.23 – diagramme « test de condition – if »

Où ConditionR1 et ConditionR2 sont deux sous-conditions de r.
On remarque que, comme prévu, un niveau supplémentaire de macro-instruction reste à développer. Il s'agit de « Test qe et until qrl ». Son diagramme est similaire dans sa structure à celui que nous avons déjà rencontré pour « Test qe et maintain ql » :

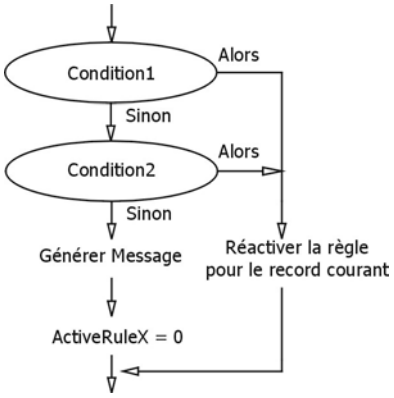


fig. 5.24 – diagramme « test de condition – qe || Until qrl »

g. Unless :

```
Fct : Unless qrm
Type : (val → bool) → (val → bool) → mod → bool
Cond. : m ≠ ∅

let rec Unless qr= function [e] → re || qe
                | e :: l → if (re) then true
                                else (qe) && (Unless qrl) ;;
```

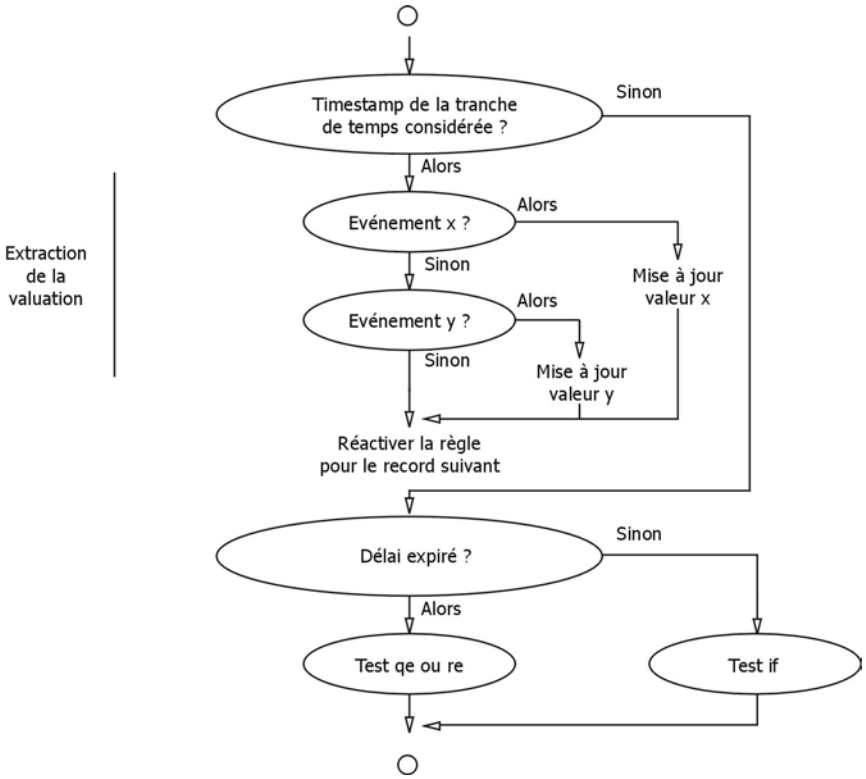


fig. 5.25 – diagramme « Unless»

Comme on peut le constater, les différences sont une fois de plus minimes entre les patterns de nature voisine. Le « Test if » a une structure similaire à celle développée pour Until. La seule différence réside dans la macro-instruction « Test qe ou re ».

Voici son diagramme :

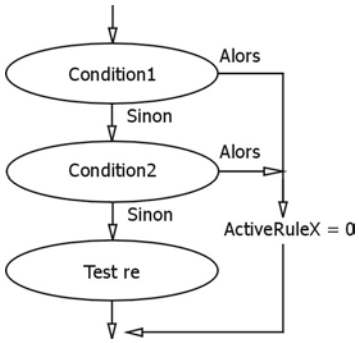


fig. 5.26 – diagramme « test de condition – qe || re »

5.5. Dernière étape de traduction : Code RUSSEL

5.5.1. Structures de données

Par soucis de complétude, nous vous présentons ici un résumé des structures de données que nous avons définies et utilisées pour la traduction. Cela peut paraître à première vue de la « cuisine interne », mais les structures décrites ci-dessous sont fortement liées au design que nous avons choisi, à savoir l'utilisation massive de règles paramétrées et l'utilisation d'une règle « supérieure » Supervisor. Cette partie nous paraît en tous cas essentielle pour celui qui désire s'attarder un peu plus sur le code Java développé.

- `TabVar :`

C'est l'ensemble des évènements considérés dans le système (ceux que l'on va contrôler). A chaque nom d'événement est associé une « valeur par défaut ». Cet ensemble sera utilisé dans init_action, comme ensemble de paramètres lors du déclenchement de la règle Supervisor. Rappelons que, en plus de cet ensemble d'éléments (nom, valeur), Supervisor reçoit comme paramètre le premier timestamp. Au sein de Supervisor, nous n'utiliserons pas les évènements véritables mais des références à ces évènements (de type ev_0, $valeur_0$, ev_1, $valeur_1$,…). En réalité, chaque règle travaille avec ses propres variables locales, ev + n° pour les noms d'événements et val ou valeur + n° pour les valeurs de ces événements. Ces variables locales ne sont instanciées que par les paramètres d'appel des différentes règles.

- `TabEx :`

C'est l'ensemble des évènements locaux à une règle RuleX. Il est utilisé pour le passage de paramètres lors du déclenchement d'une RuleX au sein de Supervisor. De ce fait, si cet ensemble contient bien les noms réels des évènements, il ne

contient pas, en revanche, les valeurs, mais bien les références aux valeurs, relatives à Supervisor. Tout comme pour Supervisor, RuleX ne travaillera pas sur des évènements véritables mais sur ses propres références aux évènements (instanciés lors du déclenchement par Supervisor).

- `SimpleCondition & SimpleCondition2 :`

Ce sont les expressions, en RUSSEL, des formules logiques simples, tels que l'événement simple, la conjonction ou la disjonction. Ces conditions étant utilisées au sein des règles, il est nécessaire de remplacer les noms des évènements par leurs références relatives à ces règles. Par exemple, soit un nom d'événement a, désigné dans une règle par ev_1, un nom d'événement b désigné par ev_2, et une valeur c. Les évènements « a=b » et « a=c » seront traduits comme « val_1=val_2 » et « val_1=c » à l'intérieur de cette règle.

- `TabCondition :`

C'est le tableau qui contiendra les formules logiques. Le premier élément renseigne sur la nature de la formule logique (condition simple, implication, équivalence), le deuxième élément est une condition simple. Dans le cas d'une implication ou d'une équivalence, une deuxième condition simple est ajoutée.

Schématiquement, à la formule logique `EQ(a=REF(b), a=c)` correspondra le tableau suivant :

EQ	(val_1=val_2) and (val_1=c)	(not(val_1=val_2)) and (not(val_1=c))

fig. 5.26 – representation de (a=b) ⇔ (a='c')

5.5.2. Syntaxe d'entrée

Il faut d'emblée faire une distinction très nette entre les patterns de logique temporelle (Achieve, Cease, Maintain, Avoid, Next, Until, Unless), et les formules de logique classique qui y sont incorporées (P-Part et Q-Part). Dans notre optique, le pattern est choisi dans une liste. Ce sont les formules de logique classique qui font l'objet d'une saisie de la part de l'utilisateur, suivant une syntaxe prédéfinie.

Cette syntaxe, assez simple, est la suivante :

- L'événement simple : e ⇒ `nom=valString` ou `/nom=valEntier`
- La conjonction : $e_1 \wedge e_2 \wedge \ldots \wedge e_n$ ⇒ `AND(`e_1`,`e_2`,…,`e_n`)`
- La disjonction : $e_1 \vee e_2 \vee \ldots \vee e_n$ ⇒ `OR(`e_1`,`e_2`,…,`e_n`)`
- L'implication : $e_1 \Rightarrow e_2$ ⇒ `IMPL(`e_1`,`e_2`)`
- L'équivalence : $e_1 \Leftrightarrow e_2$ ⇒ `EQ(`e_1`,`e_2`)`

Concernant l'événement simple, rappelons que RUSSEL ne gère que les entiers ou les chaînes de caractère. Le préfixe « / » sert à désigner une valeur entière.

Comme nous l'avons déjà évoqué, nous pouvons avoir des expressions AND dans des expressions OR, et des expressions OR dans des IMPL et des EQ. Les expressions IMPL et EQ seront par ailleurs remplacées par leurs équivalents respectifs : $(e_1 \Rightarrow e_2) \rightarrow (\neg e_1 \vee e_2)$ et $(e_1 \Leftrightarrow e_2) \rightarrow ((e_1 \wedge e_2) \vee (\neg e_1 \wedge \neg e_2))$

5.5.3. Fonctions de traduction

Nous manipulons ici des « éléments » issus des formules de logique temporelle. Ces éléments sont soit des événements, soit des conditions logiques formées à partir de ces événements, soit des formules de logique temporelles, représentées par des patterns. Les fonctions de traduction que nous définissons ont donc comme domaine de définition ces « éléments », et comme résultat du code RUSSEL, représenté par des chaînes de caractères (strings).

a. Domaines – Eléments de la syntaxe d'entrée :

Nous pouvons définir :

e ∈ EVENT :

Ensemble des événements. Ces événements peuvent être du type *string* ou du type *entier*. Ils sont caractérisés par un nom et une valeur, et se trouvent à l'intérieur des conditions.

c ∈ condition :

Ensemble des conditions de logique classique. Elles forment, à l'aide des connecteurs de logique temporelle (repris ici sous forme de patterns), des formules de logique temporelle. Ces conditions peuvent être des événements simples, des conjonctions, des disjonctions, des implications et des équivalences.

el ∈ ELEMENTS :

Désigne l'ensemble des diagrammes des patterns, ainsi que les éléments de base de ces diagrammes. Ces éléments de base sont : Extraction de valuation, Test de tranche de temps, Test spécifique selon pattern et Règle.

b. Particularités :

Pour toute règle RuleX, il y a une variable activeRuleX déclarée. Cette variable est nécessaire pour éviter d'activer continuellement la même règle lors des traitements. De ce fait, on testera la valeur de cette variable en même temps que la condition P

```
⇒ pour n règles :
  <globalVar> =
       "global internal activeRule1, activeRule2,
       … , activeRuleN : integer ;"
```

La règle init_action appelle Supervisor avec comme paramètres les éléments de TabVar.

```
⇒ pour n règles :
  <init_action> =
       "init_action ;
          begin
                activeRule1 := 0 ;
                activeRule2 := 0 ;
                …
                activeRuleN := 0 ;
                trigger off for_next Supervisor(0, <Events&Values>)
          end."
```

c. Evénements :

$\mathcal{E}$: **event → string * string * string *string**

$\mathcal{E}$ [nom=valString] = (**nom**,**'valString'**,valpos,valposx)
où valpos = **val** + position dans TabVar
valposx = **val** + position dans tabEX

$\mathcal{E}$ [/nom=valEntier] = (**nom**,**valEntier**,valpos,valposx)

d. Conditions :

$\mathcal{C}$: **condition → string * string**

$\mathcal{C}$ [nom=valString] = ("<third(e)>=<second(e)>"," ")
si on est dans Supervisor
("<fourth(e)>=<second(e)>"," ")
si on est dans RuleX

où e = $\mathcal{E}$ [nom=valString]

$\mathcal{C}$ [/nom=valEntier] = ("**bytesToInt(**<third(e)>**)=**<second(e)>"," ")
dans Supervisor
("**bytesToInt(**<fouth(e)>**)=**<second(e)>"," ")
dans RuleX

où e = $\mathcal{E}$ [/nom=valEntier]

$\mathcal{C}$ [AND(e1,e2,…,eN)] = ("**(**<e1>**) and (**<e2>**) and … and (**<eN>**)**", " ")
$\mathcal{C}$ [OR(e1,e2,…,eN)] = ("**(**<e1>**) or (**<e2>**) or … or (**<eN>**)**", " ")
$\mathcal{C}$ [IMPL(e1,e2)] = ("**not (**<e1>**)**", "<e2>")
$\mathcal{C}$ [EQ(e1,e2)] = ("**(**<e1>**) and (**<e2>**)**", "**(not (**<e1>**)) and (not (**<e2>**))**")

où <ei> = $\mathcal{C}$ [ei]

e. Eléments – patterns :

$\mathcal{T}$: **Element → string**

- **<extraction de valuation> :** voir diagramme correspondant plus haut.

```
T[ diagramme Extraction ] =
   "begin
      if   (EVENT = ev0) → val0 := VALUE ;
           (EVENT = ev1) → val1 := VALUE ;
      fi;
      trigger off for_next Supervisor[RuleX](tstp,ev0,val0,ev1,val1[,delay])
   end ;"
```

Pour RuleX, on ajoute un paramètre delay qui n'existe pas dans Supervisor.

- **<test de tranche de temps> :** voir diagramme correspondant plus haut.

```
T[ diagramme Test tt ] =
   "if     (tstp = strToInt(TIMESTP) → <extraction de valuation>
           true → <test spécifique selon pattern>
   fi"
```

- **<règle>** :

```
T[ règle ] =
   "rule Supervisor[RuleX] (tstp:integer,ev0:string,valeur0:string,
                            ev1:string,valeur1:string [,delay:integer]);
      begin
           var val0,val1 : string ;
           val0 := valeur0 ;
           val1 := valeur1 ;
           <test de tranche de temps>
      end ;"
```

- **<test spécifique par pattern> :** voir diagrammes des patterns / règles développés plus haut.

pour Supervisor, nous avons à notre disposition :

- `TabEX` pour chaque *RuleX* déclenchée par *Supervisor*. Ce sont les éléments de `TabEX` qui serviront comme paramètres des *RuleX*
- un délai associé à chaque règle, que nous notons `dX`
- une condition de déclenchement, correspondant à la partie-P des *RuleX*, que nous notons `pX`.

```
pX = first(C[ partie-P ])
```

```
𝒯[ <test spécifique Supervisor> ] =
   "if  ((activeRule1 = 0) and (<p1>) →
          begin
             activeRule1 := 1 ;
             trigger off for_current Rule1(tstp,<TabEXElems>,<d1>) ;
             trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1)
          end;
        …
        ((activeRuleN = 0) and (<pN>) →
          begin
             activeRuleN := 1 ;
             trigger off for_current RuleN(tstp,<TabEXElems>,<dN>) ;
             trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1)
          end;
        true → trigger off for_current Supervisor(tstp+1,ev0,val0,
                                                  ev1,val1)
   fi"
```

pour RuleX, voyons en détail le cas d'un ***Achieve***. Comme nous l'avons déjà mis en évidence, les autres patterns ne diffèrent pas fondamentalement les uns des autres. Il suffit donc de suivre les diagrammes proposés pour les patterns pour obtenir le code RUSSEL correspondant. Pour cette traduction, nous avons à notre disposition la condition (partie-Q), formule de logique classique contenue dans les pattern. Nous noterons `q1` et `q2` respectivement les deux parties de cette condition, à savoir `first`(𝒞 **[** `partie-Q` **]**) et `second`(𝒞 **[** `partie-Q` **]**). Il se peut que cette seconde condition soit vide.

Notons également que les 'X' doivent être remplacés par les numéros de règles.

```
𝒯[ <test spécifique RuleX> ] =
   "if
        (delay = 0) →
          if (<q1>) → activeRuleX := 0 ;
             (<q2>) → activeRuleX := 0 ;
             true → begin
                          println('but non respecté !') ;
                          activeRuleX := 0 ;
                    end
          fi ;
        true →
          if (<q1>) → activeRule1 := 0 ;
             (<q2>) → activeRule1 := 0 ;
             true → trigger off for_current RuleX(tstp+1,ev0,val0,
                                                  ev1,val1,delay-1)
          fi
    fi"
```

5.3.4. Exemple de code RUSSEL

A titre d'exemple, voici, pour un pattern Achieve :

$$\Box\,[(a = \text{'12'}) \rightarrow \Diamond_{\leq 44}\,(/b = 12)]$$

Signifiant : « Si l'attribut 'a' a la valeur (chaîne de caractère) '12', alors l'attribut b aura endéans un délai de 44 la valeur (entière) 12 ».

Ce qui donne le programme RUSSEL suivant :

```
global internal activeRule1 : integer;

rule Supervisor( tstp:integer;ev0:string;valeur0:string;
                 ev1:string;valeur1:string);
begin
   var val0,val1:string;
   val0 := valeur0;
   val1 := valeur1;
   if (tstp = strToInt(TIMESTP)) -->
          begin
             if (EVENT = ev0) --> val0 := VALUE;
                (EVENT = ev1) --> val1 := VALUE
             fi;
             trigger off for_next Supervisor(tstp,ev0,val0,ev1,val1)
          end;
       true -->
          if ((activeRule1 = 0) and (val0= '12' )) -->
                begin
                   activeRule1 := 1;
                   trigger off for_current Rule1(tstp,'b',val1,44);
                   trigger off for_current Supervisor(tstp,ev0,val0,
                                                      ev1,val1)
                end;
             true --> trigger off for_current Supervisor(tstp+1,ev0,val0,
                                                      ev1,val1)
          fi
   fi
end;

rule Rule1(tstp:integer;ev0:string;valeur0:string;delay:integer);
begin
   var val0:string;
   val0 := valeur0;
   if (tstp = strToInt(TIMESTP)) -->
          begin
             if (EVENT = ev0) --> val0 := VALUE
             fi;
             trigger off for_next Rule1(tstp,ev0,val0,delay)
          end;
       true -->
          if (delay = 0) -->
                if (bytesToInt(val0) = 12 ) --> activeRule1 :=0;
                   true -->
                      begin
                         println("Alarm !");
                         activeRule1 := 0
                      end
                fi;
             true -->
                if (bytesToInt(val0) = 12 ) --> activeRule1 :=0;
                   true --> trigger off for_current Rule1(tstp + 1,
                                                  ev0,val0,delay - 1)
                fi
          fi
   fi
end;

init_action;
   begin
      activeRule1 := 0;
      trigger off for_next Supervisor(0,'a','defaultVal',
                                      'b','defaultVal')
 end.
```

6. Exemple de synthèse et résultats expérimentaux préliminaires

Nous allons maintenant reprendre l'exemple de la mine, que nous avons déjà décortiqué lors du chapitre 2. Nous allons lui appliquer notre méthode et retirer les règles RUSSEL nécessaires à la surveillance du système visant à vérifier le modèle KAOS défini dans ce même chapitre 2.
Cette illustration est une application pas à pas de notre méthode. Toutes ces étapes sont réalisées automatiquement par notre logiciel.
Afin de faciliter notre exposé, nous ne considèrerons qu'il n'y a qu'une seule mine dans notre système. Ceci nous amènent au niveau « instance », étant donné que nous appliquons notre méthode sur un cas concret (voir point 5.3.3, consacré à la mise en route d'ASAX sur une trace que nous avons créée et aux résultats obtenus).

6.1. Enoncé

Pour rappel, voici l'énoncé du « problème »:

Considérons un système de contrôle de sécurité d'une mine. La mine est surveillée par trois détecteurs de types différents: un détecteur de niveau d'eau (water level detector - WD), un détecteur de fonctionnement/de disfonctionnement de la pompe à eau (pump failure detector - PD) et un détecteur du niveau de gaz dans la mine (gas detector - GD). WD mesure le niveau d'eau collecté dans un réservoir à l'intérieur de la mine. PD détecte le disfonctionnement de la pompe évacuant l'eau du réservoir hors de la mine. Enfin, GD mesure le niveau de gaz de la mine (du méthane dans ce cas précis). Notons que le réservoir d'eau est situé au plus bas niveau de la mine et collecte l'eau qui tombe. La pompe est utilisée pour vider le réservoir lorsqu'il dépasse un niveau maximum.

Quand le niveau d'eau mesuré par WD dépasse le niveau critique WMAX, le système de contrôle doit mette en marche la pompe à eau, et quand ce même niveau d'eau (mesuré par WD) atteint ensuite le niveau minimum WMIN, le système de contrôle arrête la pompe. Quand le niveau critique MMAX est atteint, et que PD détecte un disfonctionnement de la pompe, une alarme « danger d'inondation » est mise en marche par le système de contrôle, ce qui va provoquer l'évacuation de la mine. Quand le niveau de gaz mesuré par GD atteint le niveau critique GMAX, le système de contrôle lance une alarme « danger de suffocation », qui aura également pour effet de provoquer l'évacuation de la mine. De plus, pour éviter tout risque d'explosion, la pompe d'évacuation d'eau ne doit pas fonctionner lorsque le niveau de gaz de la mine dépasse GMAX.

Suite à l'application de méthode d'analyse proposée par KAOS, nous avons obtenu le graphe de décomposition représenté par la figure 5.1. (nous avons volontairement fait ressortir les exigences du graphe).

6.2. ORKA

Notre méthode propose, à partir des exigences, d'obtenir un ensemble de règles RUSSEL, nécessaire à la surveillance. Les spécifications de ces exigences sont présentées ci-dessous (reprises du chapitre 2, au niveau « domaine » et pour plusieurs mines).

```
Requirement Achieve[AlarmTriggered]
                        (dans le sous but Avoid[DrowningDanger])
   FormalDef : (∀ m:Mine, wd:WaterLevelDetector, a:Alarm,
                           p: Pump, dp:PumpFailureDetector)

               Observing(wd,m) ∧ Controlling(a,m)
               ∧ Observing(dp,p) ∧ (wd.level > m.MMAX)
               ∧ ¬dp.failure
               ⇒ ◊ a.active

Requirement Achieve[AlarmTriggered]
                        (dans le sous but Avoid[Suffocation])
   FormalDef : (∀ m:Mine, gd:GasLevelDetector, a:Alarm)

               Observing(gd,m) ∧ (gd.level > m.GMAX)
               ∧ Controlling(a,m)
               ⇒ ◊ a.active

Requirement Next[WaterLevelControlled]

   FormalDef : (∀ m:Mine, wd:WaterLevelDetector,
                           dp:PumpFailureDetector, p:Pump)

               Observing(wd,m) ∧ Controlling(p,m)
               ∧ Observing(dp,p) ∧ ¬dp.failure
               ∧ (wd.level >= m.WMAX)
               ⇒ o (wd.level < m.WMAX)

Requirement Achieve[PumpTurnedDown]

   FormalDef : (∀ m:Mine, gd:GasLevelDetector, p:Pump)

               Observing(gd,m) ∧ Controlling(p,m)
               ∧ (gd.level > m.GMAX)
               ⇒ ◊ ¬p.active
```

Notons que les exigences du modèle illustrant le chapitre 2 ne tiennent pas compte de la granularité du déroulement d'un scénario du système sur lequel nous travaillons. La valeur de 't' doit être déterminée par la personne qui se servira de notre logiciel. Dans le cadre de cette illustration, nous allons la fixer arbitrairement à 10.

De plus, en ce qui concerne l'exigence Next[WaterLevelControlled], la condition `wd.level = m.WMAX` a été changée en `wd.level >= m.WMAX` pour faire face à des

événements où le niveau d'eau passe de 395 à 410 unités par exemple (ce cas n'étant pas pris en compte avec le =).

Supervisor

Pour rappel, l'approche Supervisor que nous utilisons consiste à découper chaque pattern en deux morceaux appelés P-part et Q-part. Voici les décompositions des exigences que nous avons extraites:

```
Requirement Achieve[AlarmTriggered]
                        (sous but Avoid[DrowningDanger])

    P-part : (wd.level > m.MMAX) ∧ (¬dp.failure)
    Q-part : a.active


Requirement Achieve[AlarmTriggered]
                        (sous but Avoid[Suffocation])

    P-part : gd.level > m.GMAX
    Q-part : a.active


Requirement Next[WaterLevelControlled]

    P-part : (¬dp.failure) ∧ (wd.level = m.WMAX)
    Q-part : wd.level < m.WMAX

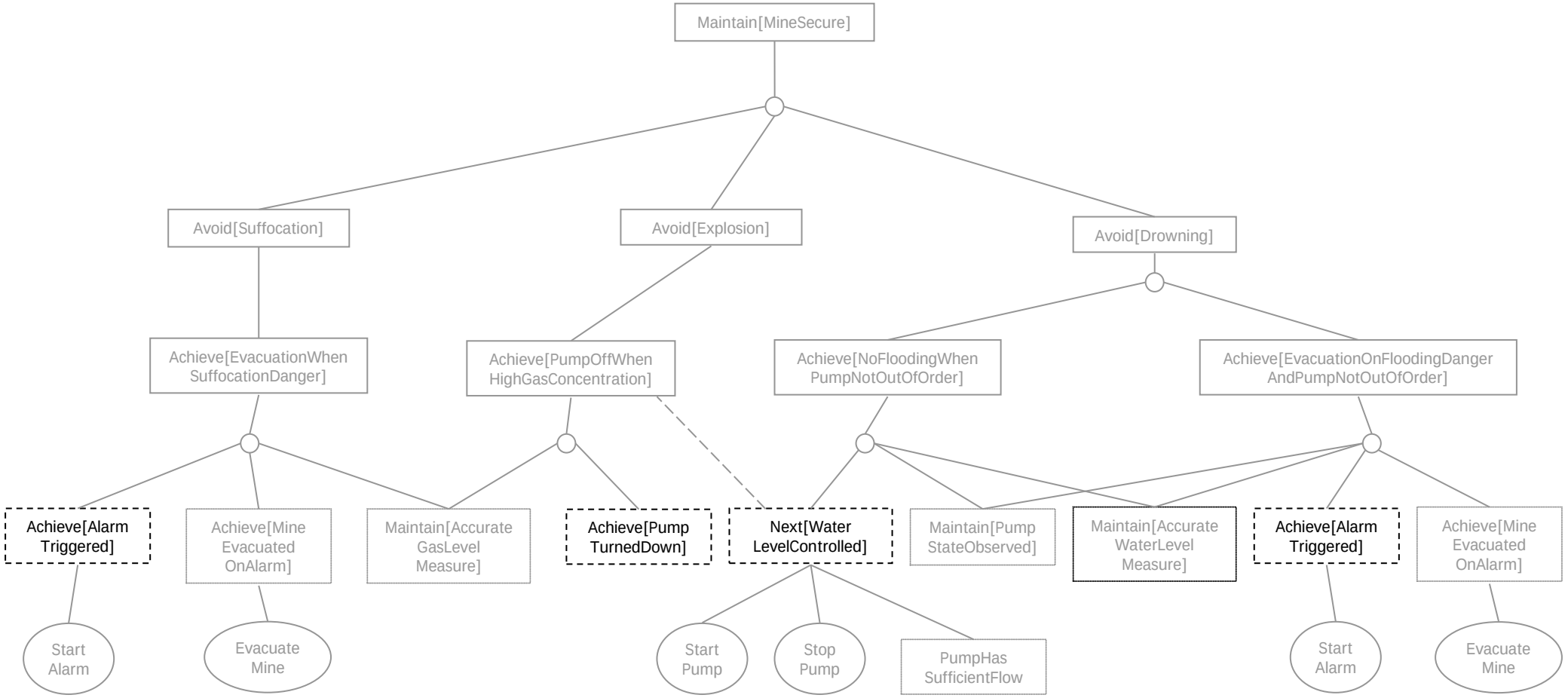

Requirement Achieve[PumpTurnedDown]

    P-part : gd.level > m.GMAX
    Q-part : ¬p.active
```

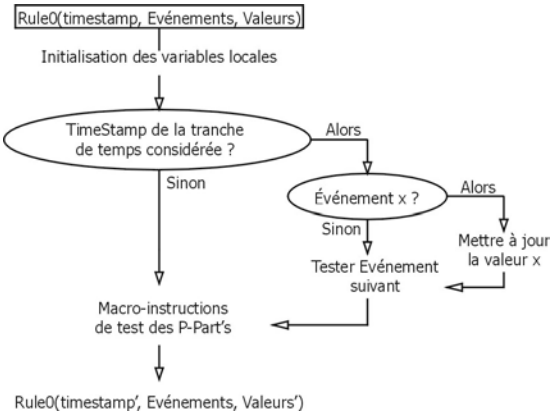
Reprenons les types des variables et constantes dont nous avons besoin pour notre vérification (les noms que nous allons leur attribuer dans le cadre de cette illustration sont indiqués entre parenthèses):

```
wd.level: integer (wdlevel)
gd.level: integer (gdlevel)
a.active: boolean (alarm)
m.MMAX: integer (mmax)
m.GMAX: integer (gmax)
m.WMAX: integer (wmax)
dp.failure: boolean (dpfail)
p.active: boolean (pump)
```

De même que pour le délai (t), nous avons fixé les valeurs des constantes wmax, mmax et gmax à 400, 600 et 430.


Maintain[MineSecure]
Avoid[Suffocation]
Avoid[Explosion]
Avoid[Drowning]
Achieve[EvacuationWhen SuffocationDanger]
Achieve[PumpOffWhen HighGasConcentration]
Achieve[NoFloodingWhen PumpNotOutOfOrder]
Achieve[EvacuationOnFloodingDanger AndPumpNotOutOfOrder]
Achieve[Alarm Triggered]
Achieve[Mine Evacuated OnAlarm]
Maintain[Accurate GasLevel Measure]
Achieve[Pump TurnedDown]
Next[Water LevelControlled]
Maintain[Pump StateObserved]
Maintain[Accurate WaterLevel Measure]
Achieve[Alarm Triggered]
Achieve[Mine Evacuated OnAlarm]
Start Alarm
Evacuate Mine
Start Pump
Stop Pump
PumpHas SufficientFlow
Start Alarm
Evacuate Mine

Nous allons donc avoir une règle Supervisor qui va, suivant les événements qui se produisent, activer une des 4 règles correspondant à la « Q-part » des 4 exigences. Pour rappel, voici l'architecture complète de la règle Supervisor:



Pour chaque tranche de temps, la règle Supervisor va recueillir les événements pertinents, en rapport avec les exigences qui nous intéressent (les valeurs des variables correspondantes seront donc mises à jour).
Ce qui donne, en RUSSEL, le code suivant:

```
rule Supervisor(tstp:integer;ev0:string;val0:integer;ev1:string;val1:string;
                ev2:string;val2:integer;ev3:string;val3:string;
                ev4:string;val4:string);
if (tstp = strToInt(TIMESTP)) -->
   begin
      if (EVENT = ev0) --> val0 := VALUE;
         (EVENT = ev1) --> val1 := VALUE;
         (EVENT = ev2) --> val2 := VALUE;
         (EVENT = ev3) --> val3 := VALUE;
         (EVENT = ev4) --> val4 := VALUE
      fi;
      trigger off for_next Supervisor(tstp,ev0,val0,ev1,val1,
                                      ev2,val2,ev3,val3,ev4,val4)
   end;

   true -->
      if ((activeRule1 = 0) and (val0 > 600 ) and (val3 = 'false' )) -->
         begin
            activeRule1 := 1;
            trigger off for_current Rule1(tstp,alarm,val1,10);
            trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1,
                                               ev2,val2,ev3,val3,ev4,val4)
         end;

         ((activeRule2 = 0) and (val2 > 430 )) -->
            begin
               activeRule2 := 1;
               trigger off for_current Rule2(tstp,alarm,val1,10);
               trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1,
                                                  ev2,val2,ev3,val3,ev4,val4)
            end;

         ((activeRule3 = 0) and (val3='false' and val0 = 400)) -->
            begin
               activeRule3 := 1;
               trigger off for_current Rule3(tstp,wdlevel,val0,1)
               trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1,
                                                  ev2,val2,ev3,val3,ev4,val4)
            end;
```

```
          ((activeRule4 = 0) and (val2 > 430 )) -->
             begin
                activeRule4 := 1;
                trigger off for_current Rule4(tstp,pump,val4,10);
                trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1,
                                                   ev2,val2,ev3,val3,ev4,val4)
             end;

          true --> trigger off for_current Supervisor(tstp+1,ev0,val0,ev1,val1,
                                                   ev2,val2,ev3,val3,ev4,val4)
       fi
fi;
```

La règle Supervisor sera appelée au début de l'exécution d'ASAX avec les paramètres suivants:

```
trigger off for_next Supervisor(0,wdlevel,'defaultVal',
                                alarm,'defaultVal',gdlevel,'defaultVal',
                                dpfail,'defaultVal',pump,'defaultVal')
```

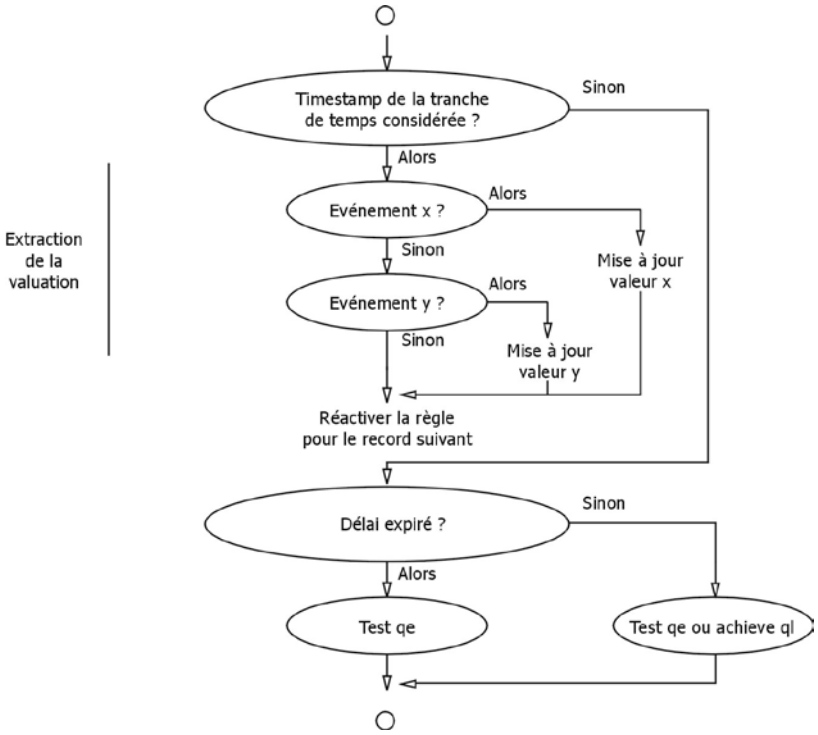
On retrouve bien ici les variables nécessaires à notre vérification (le timestamp est à 0 car on considère l'exécution la parcours de la trace depuis le temps 0 -début du scénario).

Autres règles

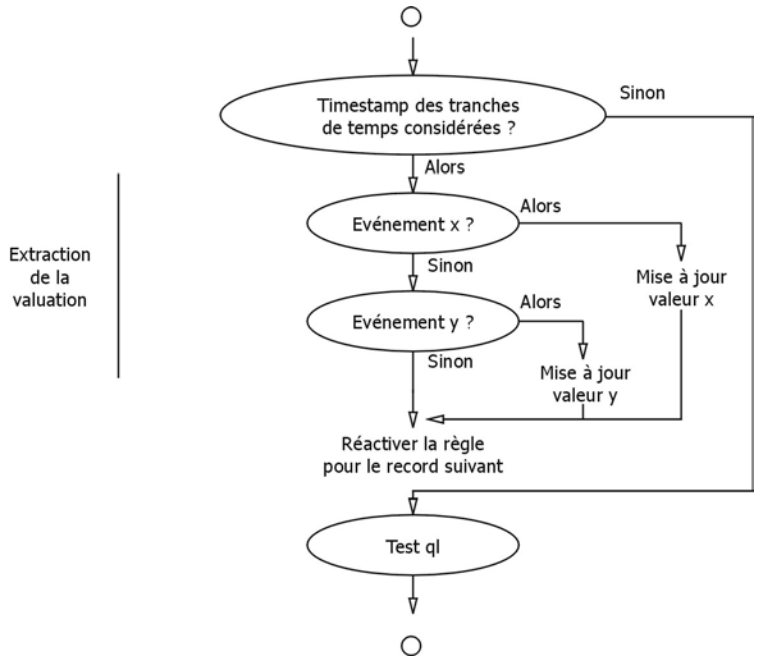
Comme nous l'avons vu plus haut, chaque exigence va correspondre à une règleX qui vérifiera la partie Q (Q-part) du pattern concerné, la partie étant gérée par Supervisor qui appellera la règleX le cas échéant.

Nous n'avons que deux patterns représentés ici (avec les diagrammes correspondants):

Achieve □ (P → ◊≤t Q)



Next □ (P→ o Q)



Requirement Achieve[AlarmTriggered]
(sous but Avoid[DrowningDanger])

Nous devons tester ici si la valeur de « alarm » est vraie endéans 10 tranches de temps.

```
rule Rule1(tstp:integer;ev0:string;val0:string;delay:integer);
if (tstp = strToInt(TIMESTP)) -->
      begin
         if (EVENT = ev0) --> val0 := VALUE
         fi;
         trigger off for_next Supervisor(tstp,ev0,val0,ev1,val1,
                                          ev2,val2,ev3,val3,ev4,val4)
      end;

   true -->
      if (delay = 0) -->
         if (val0 = 'true' ) --> activeRule1 :=0;
            true -->
               begin
                  println('Alarm !');
                  activeRule1 := 0
               end
         fi;

         true -->
         if (val0 = 'true' ) --> activeRule1 :=0;
            true --> trigger off for_current Rule1(tstp+1,
                                                   ev0,val0,delay-1)
         fi
      fi
fi;
```

Requirement Achieve[AlarmTriggered]
(sous but Avoid[Suffocation])

Ce pattern, même si la partie P est différente, subit exactement le même traitement que le précédent, vu qu'ici, nous ne travaillons que sur la partie Q. Nous obtenons exactement le même code que pour la exigence précédente.

Requirement Next[WaterLevelControlled]

```
rule Rule3(tstp:integer;ev0:string;val0:string;delay:integer);
if (tstp >= (strToInt(TIMESTP) - delay)) -->
      begin
         if (EVENT = ev0) --> val0 := VALUE
         fi;
         trigger off for_next Supervisor(tstp,ev0,val0,ev1,val1,
                                          ev2,val2,ev3,val3,ev4,val4)
      end;
   true -->
      if (delay = 0) -->
         if (val0 < 40 ) --> activeRule3 :=0;
            true -->
               begin
                  println('Alarm !');
                  activeRule3 := 0
               end
         fi;

         true -->
         if (val0 < 40 ) --> activeRule3 :=0;
            true --> trigger off for_currentRule3(tstp+1,
                                                  ev0,val0,delay-1)
         fi
      fi
fi;
```

Requirement Achieve[PumpTurnedDown]

Cette exigence se voit traitée comme les deux premières, pour la partie Q, si ce n'est que l'on attend la valeur « false ».

```
rule Rule4(tstp:integer;ev0:string;val0:string;delay:integer);
if (tstp = strToInt(TIMESTP)) -->
      begin
         if (EVENT = ev0) --> val0 := VALUE
         fi;
         trigger off for_next Supervisor(tstp,ev0,val0,ev1,val1,
                                          ev2,val2,ev3,val3,ev4,val4)
      end;

   true -->
      if (delay = 0) -->
         if (val0 = 'false' ) --> activeRule4 :=0;
            true -->
               begin
                  println('Alarm!');
                  activeRule4 := 0
               end
         fi;
         true -->
         if (val0 = 'false' ) --> activeRule4 :=0;
            true --> trigger off for_current Rule4(tstp+1,
                                                   ev0,val0,delay-1)
```

```
         fi
      fi
fi;
```

Init_action

N'oublions pas la règle de départ et les définitions de nos variables globales:

```
global internal activeRule1 : integer;
global internal activeRule2 : integer;
global internal activeRule3 : integer;
global internal activeRule4 : integer;

init_action;
   begin
      activeRule1 := 0;
      activeRule2 := 0;
      activeRule3 := 0;
      activeRule4 := 0;
      trigger off for_next Supervisor(0,wdlevel,'defaultVal',
         alarm,'defaultVal',gdlevel,'defaultVal',dpfail,'defaultVal',
         pump,'defaultVal')
   end;
```

6.3. Utilisation de notre logiciel

Voici, pour chaque exigence, ce que nous introduisons dans le logiciel que nous avons créé, avec la syntaxe définie plus haut:

Requirement Achieve[AlarmTriggered]
(sous but Avoid[DrowningDanger])

□ **[AND(/wdlevel > 600,dpfail = 'false') → ◊≤ 10 (alarm = 'true')]**

Requirement Achieve[AlarmTriggered]
(sous but Avoid[Suffocation])

□ **[(/gdlevel > 430) → ◊≤ 10 (alarm = 'true')]**

Requirement Next[WaterLevelControlled]

□ **[AND(dpfail = 'false', /wdlevel >= 400) → o (/wdlevel < 400)]**

Requirement Achieve[PumpTurnedDown]

□ **[(/gdlevel > 430) → ◊≤ 10 (pump = 'false')]**

Le code obtenu suite à l'utilisation de notre petite application graphique est très similaire à celui proposé au point précédent. Les seules différences proviennent d'une automatisation des procédures de transformation des exigences qui sont introduites une à une. Le code complet peut être consulté en annexe.

6.4. Résultats expérimentaux

6.4.1. Jeu de test

A partir du code obtenu pour l'illustration de la mine, nous reprenons le code généré par ORKA pour faire quelques tests de performance. Ainsi, nous générons une trace de 100 éléments pertinents pour tester notre méthodologie.

En plus des événements attendus (comme « wdlevel » ou encore « pump »), nous ajoutons des événements de l'environement comme le nombre de mineurs dans la mine (« miners_inside », entier) et un témoin du fait que les mineurs creusent (« miners_digging », booléen). Nous plaçons ces événements tous les 20 et 100 records respectivement. Notons que lorsque l'alarme est déclenchée, le nombre de mineurs tombe à 0 et ils ne creusent plus (ceci correspondant à l'hypothèse Achieve[MineEvacuatedOnAlarm]).

Sur les 100 éléments de notre trace, que se passe-t-il dans le système?

(1) Le niveau de gaz passe atteint exactement la valeur GDMAX, l'alarme est déclenchée, la pompe coupée, mais trop tard.

(2) L'alarme est stoppée un peu plus tard, et le niveau de gaz redescend peu à peu (mais lentement, donc la pompe n'est toujours pas réactivée)

(3) Conséquence de la coupure de la pompe, le niveau d'eau monte, et donc passe la barre de WMAX et continue à monter et passe ensuite la valeur MMAX; l'alarme est déclenchée.

(4) Le niveau de gaz étant redescendu, la pompe est réactivée manuellement et le niveau d'eau baisse rapidement. L'alarme est stoppée un peu plus tard.

(5) Jouant de malchance, la mine voit son niveau d'eau augmenter rapidement, suite à une faille non loin d'une source. Le niveau d'eau dépasse les valeurs WMAX puis MMAX. L'alarme est déclenchée.

(6) En même temps que les événements décrits au point (5), le niveau de gaz monte suite à un coup de « grisou ». Ceci arrive juste après que le niveau d'eau ait dépassé MMAX. L'alarme étant déclenchée par le niveau d'eau trop élevé, cet événement est correctement pris en compte et, de plus, la pompe est coupée dans les temps.

(7) Retour à la normale pour la mine : alarme désactivée, pompe réactivée, et les seuils (WMAX, MMAX et GMAX) sont à nouveau respectés.

Reprenons les événements, et faisons-les correspondre aux événements dans la trace et aux types de tests que nous désirons effectuer :

(1) Le niveau de gaz (`gdlevel`) passe atteint exactement la valeur GDMAX (`430`), l'alarme est déclenchée en t+9 (`alarm='true'`), la pompe coupée en t+10 (`pump='false'`)

→ *test du pattern Achieve[AlarmTriggered] (sous but Avoid[Suffocation]) ;*
→ *test des extrémités pour la durée d'activation de la trace.*

(2) L'alarme est stoppée un peu plus tard (`alarm='false'`), et le niveau de gaz redescend peu à peu (mais lentement, donc la pompe n'est toujours pas réactivée)

→ *pas de test, retour à la normale.*

(3) Conséquence de la coupure de la pompe, le niveau d'eau (`wdlevel`) monte, et donc passe la barre de WMAX (`400`) et continue à monter et passe ensuite la valeur MMAX (`600`) ; l'alarme est déclenchée en t+8 (`alarm='true'`).

→ *test des patterns Next[WaterLevelControlled] et Achieve[AlarmTriggered] (sous but Avoid[DrowningDanger]).*

(4) Le niveau de gaz étant redescendu (`<430`), la pompe est réactivée manuellement (`pump='true'`) et le niveau d'eau baisse rapidement (`<400`). L'alarme est stoppée un peu plus tard (`alarm='true'`).

→ *pas de test, retour à la normale.*

(5) Jouant de malchance, la mine voit son niveau d'eau (`wdlevel`) augmenter rapidement, suite à une faille non loin d'une source. Le niveau d'eau dépasse les valeurs WMAX (`400`) puis MMAX (`600`). L'alarme est déclenchée en t+4 (`alarm='true'`).

→ *test du pattern Achieve[AlarmTriggered] (sous but Avoid[DrowningDanger]), avec « mort » de la règle.*

(6) En même temps que les événements décrits au point (5), le niveau de gaz (`gdlevel`) monte suite à un coup de « grisou ». Ceci arrive juste après que le niveau d'eau ait dépassé MMAX (`600`). L'alarme étant déclenchée par le niveau d'eau trop élevé (`>600`), cet événement est correctement pris en compte et, de plus, la pompe est coupée dans les temps, en t+2 (`pump='false'`).

→ *test des patterns Achieve[AlarmTriggered] (sous but Avoid[Suffocation]) et Achieve[PumpTurnedDown], avec « mort » des règles (la « mort » de la première étant due à l'activation de l'alarme au point (5)).*

(7) Retour à la normale pour la mine : alarme désactivée (`alarm='false'`), pompe réactivée (`pump='true'`), et les seuils (WMAX, MMAX et GMAX) sont à nouveau respectés (`gdlevel<430, wdlevel<400`).

→ *pas de test, retour à la normale.*

Les événements de notre trace sont générés « à la main » dans un simple fichier texte et sont au format suivant :

```
<trace> ::= <record> eoln (<record> eoln)
<record> ::= <timestamp>;<evenement>&<valeur>
<timestamp> ::= entier
<evenement> ::= string
<valeur> ::= string | entier
```

Le code de l'adaptateur de format ainsi que la trace dans son intégralité sont disponibles en annexe.

6.4.2. Résultats

Nous lançons ASAX avec la trace définie plus haut et voici ce que nous obtenons (sur une machine Intel Pentium III 600 Mhz équipée de 128 Mo de mémoire vive, avec Solaris 2.7 et la version monoposte d'ASAX installée) :

```
begin parsing description file ...
         asax :     code_bytes.asa
Processing audit trail ...

Alarm ! - Obstacle detected in Requirement Achieve[PumpTurnedDown] (11)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (43)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (44)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (45)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (46)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (47)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (48)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (50)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (51)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (52)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (53)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (54)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (56)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (57)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (58)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (59)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (68)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (69)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (70)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (72)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (73)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (74)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (76)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (78)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (79)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (80)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (82)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (84)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (86)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (88)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (90)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (92)

end of audit trail reached.
Processing completion rules ...

End of emulation.

user time div HZ : 0.010000
system time div HZ: 0.010000
total time div HZ : 0.020000
```

Parfait, les alarmes sont générées pour les patterns aux moments où elles sont attendues (nous avons également indiqué les « timestamps » entre parenthèses). Une version des résultats pour cet exemple avec les notifications des « morts » des règles est disponible en annexe.

Notre approche ne souffre pas de défaut de conception. Voyons plus loin maintenant.

6.4.3. Tests approfondis et résultats

Poussons plus loin donc, et parlons en termes de performance. Pour ce faire, nous reprenons la trace décrite ci-dessus et la produisons pour obtenir des traces de 1000, 3000, 5000, 10000 et 100000 éléments respectivement.

Trace length	**1000**	**3000**	**5000**	**10000**	**100000**
	0.15	0.45	0.81	1.88	75.27

fig. 6.1 - Temps d'exécution (en secondes) pour les 4 formules

Il semble que notre exécution rencontre quelques difficultés dès que l'on aborde les 100000 éléments (beaucoup plus de temps pour achever la trace).

6.4.4. Tests sur d'autres types de traces et résultats

Pour la rédaction d'un papier destiné à un « workshop » de la NASA à Copenhagen en juillet 2002[22], nous avons, suite à la lecture de [HaRo01] et [FiSi01], décidé de comparer notre approche avec celles proposées par les papiers au point de vue des performances.

Les deux approches proposées par [HaRo01] et [FiSi01] sont des algorithmes écrits pour tester une seulle formule temporelle sur des traces de tailles finies.

Les auteurs de [HaRo01] proposent d'utiliser des automates, en traversant ceux-ci suivant trois méthodes : en profondeur d'abord, en largeur d'abord et en arrière. La troisième méthode est celle qui s'est avérée la plus conluante, et que nous avons retenue pour notre comparatif. L'implémentation de la méthode a été faite en Java. Les tests ont été réalisés sur un laptop VAIO Sony équipé d'un Pentium III 850Mhz, sur lequel tournait Linux RedHat v7.0 et Sun JDK1.3.1. Notons que cette approche ne fonctionne que pour des formules temporelles avec des opérateurs futurs.

Les formules testées sont :

- □ ◊ z traduite dans notre méthodologie en :
  □ [true → ◊ z='true']23
- □ ◊ a traduite dans notre méthodologie en :
  □ [true → ◊ a='true']
- □ [b → ¬ a U a] traduite dans notre méthodologie en :

[22] RV'02 - Second Workshop on *Runtime Verification* (http://ase.arc.nasa.gov/rv2002/)
[23] le **true** pour la partie P de la formule est interpretée comme un appel sans condition à la règle RuleX correspondant à la partie Q

□ [b='true'→ NOT(a='true') U a='true']

La trace utilisée pour les tests est constituée de 10% de « a »[24] (a='true', 40% de « b », 25% de « c » and 25% de « d », avec un « z » ajouté à la fin de la trace pour le premier test, et un « a » pour les deux autres (afin de rendre le Achieve vrai).

Notons que le délai chez nous est fixé à la taille de la trace.

Formules et approaches		Longueur de trace				
		1000	**3000**	**5000**	**10000**	**100000**
□ ◊ z	**ORKA**	0.010	0.010	0.010	0.100	0.100
	[FS01]	0.023	0.036	0.055	n/a	n/a
□ ◊ a	**KAOS on ASAX**	0.030	0.100	0.170	0.350	6.510
	[FS01]	0.024	0.036	0.056	n/a	n/a

fig. 6.2 – Temps d'exécution (en secondes) pour □ ◊ z and □ ◊ a

Trace length	**1000**	**3000**	**5000**	**10000**	**100000**
	0.04	0.13	0.22	0.47	12.26

fig. 6.3 – Temps d'exécution (en secondes) pour □ (b → a U a)

Les auteurs de [HaRo01] proposent d'utiliser une méthode de réécriture des formules, appelée Maude. Leur méthode s'applique sur des formules utilisant aussi bien des opérateurs sur le futur que sur le passé. L'implémentation est faite an Java, et s'effectue dans le cadre des recherches sur Java PathExplorer (JPaX). Les tests ont été réalisés sur un PC équipé d'un processeur tournant à 1.7Ghz avec 1Go de mémoire vive (le système d'exploitation n'était pas précisé).

La formule testée est **□ [b → ◊ a],** traduite dans notre méthodologie en **□ [b='true'→ ◊ a='true']**

La trace utilisée est une répétition des événements « a b, a, c a, a b, c b, a b, a, c a, a b, c b »[25].

Formule et approaches		Longueur de trace				
		1000	**3000**	**5000**	**10000**	**100000**
□(a → ◊ b)	**ORKA**	0.07	0.22	0.38	0.85	21.19
	[HS01]	quelques millisecondes			(. < 1)	. > 3

fig.6.4 – Temps d'exécution (en secondes) pour □ (a → ◊ b)

Nous sommes encore allés plus loin, en regroupant les 4 formules et en lançant ASAX (en gardant la dernière trace, plus compliquée à gérer) :

Longueur de trace	**1000**	**3000**	**5000**	**10000**	**100000**

[24] `tstp;a&true` pour nous (même type pour les « b », « c », « d » et « z »)
[25] devenant
`0;a&true;b&true`
`1;a&true`
etc.

	0.13	0.43	0.77	1.85	about 80

fig. 6.5 - Temps d'exécution (en secondes) pour les 4 formules

Comme on peut le voir, la méthode utilisée pour traiter la trace et l'adaptation de notre méthodologie ne donnent pas des résultats très concluants. S'il est vrai qu'il a fallu retravailler légèrement certaines formules pour obtenir « nos » patterns, et que nos tests sont effectués sur une machine moins performante, les traces contenant 100000 éléments sont très difficiles à gérer.

Bien qu'ASAX ait peut-être eu quelques difficulté à gérer des traces très longues, ceci est en grosse partie du à notre traduction d'une trace native en trace NADF. En effet, selon notre apporche, si, pour la trace, on a :

tstp	ev1	Val1	ev2	val2

fig. 6.6 - Trace simple

Notre adaptateur génère :

tstp	ev1	val1

tstp	ev1	val1

fig. 6.7 - Trace NADF obtenue

Ceci devient bcp plus ennuyeux lorsque la trace de base contient 100000 records de 2 événements par timestamp, la trace NADF faisant alors 200000 records. ASAX étant très rapide, il doit tout de même travailler sur une trace deux fois plus longue.

6.4.5. Enseignements tirés de l'expérimentation

Notre approche se révèle très correcte et remplit son rôle de vérificateur à merveille, permettant de tout de suite mettre le doigt sur l'exigence non respectée par le système dont la trace a été analysée.

Malgré cela, il est assez embarassant de travailler avec une trace de plus de 100000 événements. Le fait d'avoir choisi un adaptateur de format simple (le fichier DDF ne comportant que trois éléments, pour rappel), facile à utiliser, a des conséquences qui se font ressentir dans les tests de performance.

7. Conclusion, réflexions et travaux futurs

Comme nous l'avons évoqué déjà dans l'introduction, c'est à un besoin réel et croissant que nous avons tenté de répondre. Le besoin d'un outil de vérification, de validation d'une implémentation, qui soit à la fois simple à mettre en œuvre et compatible avec des contraintes, notamment de délais, induites par l'usage de nouvelles pratiques de programmation.

C'est donc dans la lignée d'un nouveau domaine, le « Dynamic Program Monitoring », que s'inscrit notre réflexion, matérialisée par le présent mémoire.

ORKA est un outil permettant de répondre à ce besoin de vérification. Basé sur une méthode connue et qui a fait ses preuves, KAOS, et sur un outil puissant, fiable et efficace, ASAX, il permet de traduire les spécifications « comportementales » définies dans le premier en un ensemble de règles de vérification utilisable par le second. C'est le trait d'union entre une méthode de spécification, et un outil d'analyse séquentielle de fichier.

Grâce à ORKA, à KAOS et à ASAX, nous pouvons d'ores et déjà modéliser un grand nombre de comportement, vérifier un grand nombre de systèmes, et, profitant de l'efficacité d'ASAX, facilement étendre nos travaux à des systèmes informatiques de grande taille.

Cependant, cette efficacité n'a encore pu être obtenue que pour certains types de spécification KAOS. C'est le premier point qui nous semble encore à « explorer ». Nous devons enrichir le sous-ensemble de logique temporelle que nous utilisons, en incluant plus de patterns, en augmentant le degré d'imbrications des formules temporelles, en considérant les opérateurs passés, malgré le fait qu'ASAX n'implémente ni backtracking ni base de données. Le problème des instances multiples d'un même objet doit également, lors de travaux futurs, être discuté et réglé.

D'un point de vue plus « utilisateur », il est sans doute également utile d'améliorer l'interface d'ORKA, en utilisant par exemple un langage d'entrée plus proche de la logique temporelle, plutôt qu'un « formulaire » à remplir. Cette amélioration sous-entend bien sûr également une plus grande robustesse face aux erreurs de saisie, une meilleure communication des résultats et de l'avancement de la tâche.

Enfin, d'un point de vue purement technique, nous devons aussi nous attarder sur la trace d'exécution, sur sa structure et sa génération. Le format NADF permet de grouper des événements dans un même record. Ceci pourrait nous dispenser de toute la partie « extraction de valuation », mais nous obligerait à générer un adaptateur de format spécifique à chaque système rencontré, avec les noms d'événements et les types spécifiques au système. Cette plus grande complexité peut cependant apporter plus de richesse et de performance lors de l'analyse proprement dite.

Toujours concernant cette trace d'exécution, le problème de la génération de la trace reste en suspend. Nous n'avons jusqu'à présent travaillé que sur des traces

générées « à la main », ou du moins automatiquement, mais ne correspondant pas à un système réel en cours d'exécution. Une solution possible est bien entendu d'intégrer au système à vérifier un outil de génération de trace d'exécution, comme c'est le cas pour les traces d'audit des systèmes d'exploitation. Une autre solution serait de créer un outil indépendant chargé de capturer ces événements.

En bref, nous pouvons donc dire que ORKA est une première étape, démontrant la validité d'une telle approche en « Dynamic Program Monitoring », démontrant qu'il n'est pas nécessaire de passer par des mécanismes longs, complexes, laborieux et en somme peu performants, pour valider une implémentation. De par sa relative simplicité, et surtout de par son efficacité, ORKA n'aura, nous l'espérons, aucun mal à s'intégrer dans ces nouvelles pratiques de programmations, comme un outil à par entière, garant d'une certaine qualité.

Bibliographie

[BuLi91] A. Burns and A.M. Lister, “A framework for building dependable systems”. *Comp J.*, 34(2), 173-181, 1991.

[DVLF93] A. Dardenne, A. van Lamsweerde and S. Fickas, “Goal-Directed Requirements Acquisition”, *Science of ComputerProgramming*, Vol. 20, 1993, 3-50.

[DYP98] E. Dubois, E. Yu, M. Petit, “From Early to Late Formal Requirements, a Process Control Case Study”, in Proceedings of the *9th International Workshop on Software Specification and Design*, Ise-Shima (Japan), April 1998.

[FiSi01] B. Finkbeiner and H. Sipma. “Checking Finite Traces using Alternating Automata”, *Workshop on Runtime Verification*, CAV'01, 2001.

[HLCM94] N. Habra, B. Le Charlier, and A. Mounji, “Advanced Security Audit Trail Analysis on Unix (ASAX also called SAT-X). Implementation Design of the NADF Evaluator”, *Tech. report*, Institut d’informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgium, September 1994.

[HLCMM92] N. Habra, B. Le Charlier, A. Mounji and I. Mathieu, “ASAX : Software Architecture and Rule-Based Language for Universal Audit Trail Analysis”, *European Symposium on Research in Computer Security (ESORICS’92)*, Toulouse, France, Springer-Verlag, november 1992.

[HaRo01] K. Havelund and G. Rosu, “Monitoring Programs using Rewriting”, *Automated Software Engineering 2001 (ASE'01)*, San Diego, California, 26-29 November 2001, IEEE Computer Society.

[LAS93] Report of the Inquiry Into the London Ambulance Service, February 1993, The Communications Directorate, South West Thames Regional Authority, ISBN 0-905133-70-6.

[LCMS95] B. Le Charlier, A. Mounji and M. Swimmer, “Dynamic Detection and Classification of Computer Viruses Using General Behaviour Patterns”, *In Proceedings of Fifth International Virus Bulletin Conference*, Boston, Septembre 20-22, 1995.

[MaHa92] B.P. Mahony and I.J. Hayes, “A case-study in timed refinement: a mine pump”. *IEEE*
Trans. On Softw.Eng., 18(9), 817-826, 1992.

[MaPn92] Z. Manna, and A. Pnueli, “The Temporal Logic of Reactive and Concurrent Systems”, Springer-Verlag, 1992.

[MaSteP96] Z. Manna and the STep Group, “STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems”, *Proc.CAV’96 - 8th Intl. Conf. on Computer-Aided Verification*, LNCS 1102, Springer-Verlag, July 1996, 415-418.

[Math96] J. Mathai (eds), “Real-time Systems. Specification, Verification and Analysis”, C.A.R Hoare Series Editor, Prentice Hall, 1996.

[MaVa01] Xavier Martin and Nicolas Vanderavero, “Les Systèmes de Détection d’Intrusion - Leçons à tirer d’un déploiement d’ASAX Distribué dans un Contexte Universitaire”, Master’s Thesis, Institut d’Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgium, 2001.

[MLC96] A. Mounji and B. Le Charlier, “Detecting Breaches in Computer Security: A Pragmatic System with a Logic Programming Flavor”, *In Proceedings of the*

Eight Benelux Workshop on Logic Programming, Louvain-La-Neuve, Belgium, September, 1996.

[MLC97] A. Mounji and B. Le Charlier, "Continuous Assessment of a Unix Configuration: Integrating Intrusion Detection and Configuration Analysis", *In Proceedings of the ISOC' 97 Symposium on Network and Distributed System Security*, San Diego, California, 1997.

[MLCZH95] A. Mounji, B. Le Charlier, D. Zampuniéris and N. Habra, "Distributed Audit Trail Analysis", *In Proceedings of the ISOC '95 Symposium on Network and Distributed Systems Security*, San Diego, California, February 1995.

[Mou97] Abdelaziz Mounji, "Language and Tools for Rule-Based Distributed Intrusion Detection", Ph. D. Thesis, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix Namur, Belgium, September 1997.

[ORS95] S. Owre, J. Rushby and N. Shankar, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS", *IEEE Transactions on Software Engineering*, Feb. 95, Vol. 21 No. 2, 107-125.

[Pons98] Ch. Ponsard, "Analysing Composite Systems Using KAOS : The Mine Case Study", Unité d'Informatique, Université de Louvain-la-Neuve, juillet 1998.

[VLaLe98] A. van Lamsweerde and E. Letier, "Integrating Obstacles in Goal-Driven Requirements Engineering", *Proc. ICSE-98 : 20th Intrnational Conference on Software Enginering*, Kyoto, April 1998.

[VLaLe00] A. van Lamsweerde and E. Letier "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, Vol. 26 No. 10, October 2000, 978-1005.

[VLDD91] A. van Lamsweerde, A. Dardenne and F Dubisy, "KAOS Knowledge Representations as Initial Support for Formal Specification Processes", *Report RR-91-8*, Unité d'Informatique, Université de Louvain, 1991.

[VLDDD91] A. van Lamsweerde, A. Dardenne, B. Delcourt and F. Dubisy, "The KAOS Project: Knowledge Acquisition in Automated Specification of Software", *Proceedings AAAI Spring Symposium Series*, Stanford University, American Association for Artificial Intelligence, March 1991, pp. 59-62.

[VLDL98] A. van Lamsweerde, R. Darimont and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Trans. on Sofware. Engineering*, Special Issue on Inconsistency Management in Software Development, November 1998.

[VLDM95] A. van Lamsweerde, A., Darimont, R. and Massonet, P., "Goal-Directed Elaboration of Requirements for a Meeting Scheduler : Problems and Lessons Learned", *Proc. RE'95 - 2nd Int.Symp. on Requirements Engineering*, York, IEEE, 1995.

[BGLC02] Brohez, S., Grégoire, Y. and Le Charlier, B., "Obstacle Recognition with KAOS: ORKA, an implementation based on ASAX", *submitted to RV'02 - Second Workshop on* Runtime Verification *(http://ase.arc.nasa.gov/rv2002/)*

Annexes

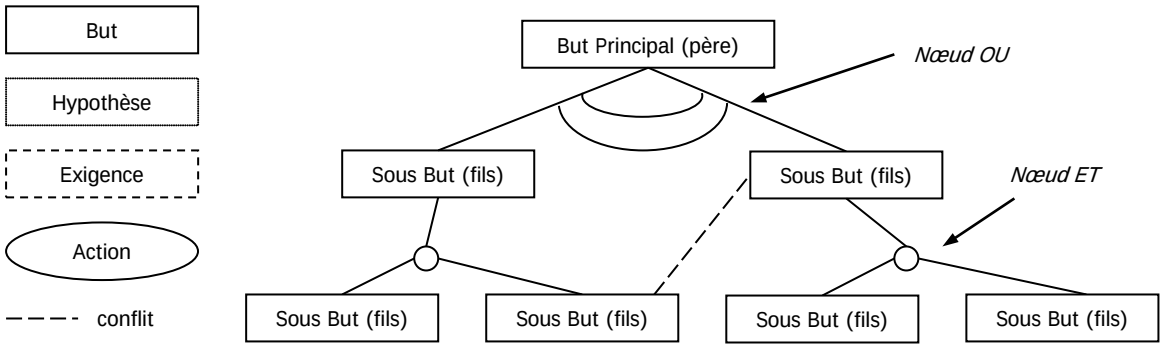
A.1. Syntaxe concrète de RUSSEL

```
<module> ::= <usage list> <global var decl> <module body>.
<module body> ::= <rule decl list>
                  | <rule decl list;> <init action> <rule decl
                  list>
<usage list> ::= <empty>
                  | <module usage> ; ... ; <module usage> ;
<module usage> ::= uses <list of modules>
<list of modules> ::= <module name>, ... , <module name>
<module name> ::= <identifier>
<global var decl> ::= <empty>
                  | <global var group>; ... ;<global var group>;
<global var group> ::= global <variable class>
<variable name>, ... ,<variable name> : <type>
<variable class> ::= <empty>
                  | internal
                  | external
<rule decl list;> ::= <empty>
                  | <rule decl list;> <rule declaration> ;
<rule decl list> ::= <empty>
                  | <rule decl list;> <rule declaration>
<rule declaration> ::= <rule heading> ; <variable declaration
                  part> <action part>
<rule heading> ::= <rule class> rule <rule name> <parameter part>
<rule class> ::= <empty>
                  | internal
                  | external
<rule name> ::= <identifier>
<parameter part> ::= <empty>
                  | (<parameter group>; ...;<parameter group>)
<parameter group> ::= <parameter name>, ... , <parameter name> :
                  <type>
<parameter name> ::= <identifier>
<type> ::= string | integer
<variable declaration part> ::= <empty>
                  | var <variable declaration>; ... ;
                  <variable declaration>;
<variable declaration> ::= <variable name>, ... ,<variable name>
                  : <type>
<variable name> ::= <identifier>
<action part> ::= skip
                  | <conditional action>
                  | <repetitive action>
                  | <compound action>
                  | <rule triggering>
<action> ::= skip
                  | <assignment>
                  | <conditional action>
                  | <repetitive action>
                  | <compound action>
                  | <rule triggering>
```

```
                  | <routine call>
<assignment> ::= <variable name> := <expression>
<keyword> ::= <identifier>
<expression> ::= <term>
                  | - <term>
                  | <expression> <additive operator> <term>
<additive operator> ::= +
                  | -
<term> ::= <factor>
                  | <term> <multiplicative operator> <factor>
<multiplicative operator> ::= _
                  | mod
                  | div
<factor> ::= <constant>
                  | <field name>
                  | <variable name>
                  | <parameter name>
                  | (<expression>)
                  | <routine call>
                  | <keyword>
<field name> ::= <identifier>
<conditional action> ::= if <guarded action>; ... ; <guarded
                  action> fi
<guarded action> ::= <condition> --> <action>
<condition> ::= <conjunction>
                  | <condition> or <conjunction>
<conjunction> ::= <literal>
                  | <conjunction> and <literal>
<literal> ::= <atom>
                  | not <atom>
<atom> ::= true
                  | false
                  | present <field name>
                  | <expression> <relational operator>
                  <expression>
                  | <field name> in <keyword>
                  | ( <condition> )
<relational operator> ::= <
                  | >
                  | <>
                  | <=
                  | >=
                  | =
<repetitive action> ::= do <guarded action>; ... ; <guarded
                  action> od
<compound action> ::= begin <action>; ... ; <action> end
<rule triggering> ::= trigger off <triggering mode> <rule call>
<triggering mode> ::= for next
                  | for current
                  | at completion
<routine call> ::= <routine name>
                  | <routine name>
                  | (<expression>,...,<expression>)
<routine name> ::= <identifier>
<rule call> ::= <rule name> (<expression>,..., <expression>)
<rule name> ::= <identifier>
```

```
<init action> ::= init action; <variable declaration part>
                  <action part>
```

A.2. Syntaxe graphique des graphes de décomposition KAOS



A.3. Implémentation de l'adaptateur de format

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>

#define OK               0
#define VBADOPEN        -1
#define VBADREAD        -2
#define VBADCLOSE       -3
#define VEOF            -4
#define FILERR          -5 /*  FILE ERRor : file has not valid KAOS trace form */
#define TCAPOVFL        -6 /*  Table CAPacity OVerFLow  */
#define TABLEN          30

typedef struct nadf_rec{
    int length;
    u_short tstp_id;
    u_short tstp_lg;
    u_long timestamp;
    u_short ev_id;
    u_short ev_lg;
    char event[16];
    u_short val_id;
    u_short val_lg;
    char value[16];
}evt;


static evt event_table[TABLEN];   /* table of events in NADF record-like form */
static int etlen;                 /* length of the table */



/* kfread reads a line of the file pointed by fp. It transforms it into as many
     records
 * as events contained in the current line (reminder : a KAOS line contains all the
 * events for a given timestamp (1st "field" of the line).
 * After reading the line, the table event_table contains every record for the
     current
 * timestamp. Each element of the table is a NADF record-like structure.
```

```
 */
int kfread(FILE *fp)
{
    int i,j;
    char t[10];
    char ev[16];
    char val[16];
    char c='\n';
    int eoln = 0;
    u_long tmstp;

    etlen = 0;

    /* reading timestamp */

    /* 1rst char : it is possible that a file is not a KAOS trace. We check it
     here.
     * We will accept files with more than 1 carriage return and/or blanks between
     records
     */

    while ((c == '\n')||(c == ' ')) {c = fgetc(fp);}
    if (c == EOF) return VEOF;
    if (c == ';') return FILERR;
    switch (c) {
    case ('1'): break;
    case ('2'): break;
    case ('3'): break;
    case ('4'): break;
    case ('5'): break;
    case ('6'): break;
    case ('7'): break;
    case ('8'): break;
    case ('9'): break;
    case ('0'): break;
    default: return FILERR;
    }
    t[0] = c;

    /* rest of timestamp */

    i = 1;
    while ((c=fgetc(fp)) != ';'){
     if ((c == EOF)||(c == '\n')) return FILERR;
     t[i] = c;
     i++;
    }
    t[i] = '\0';

    tmstp = atol(t);

    /* reading the events and values */

    j = 0;
    while (eoln == 0) {
     i = 0;
     while ((c=fgetc(fp)) != '&'){
         if ((c == EOF)||(c =='\n')) return FILERR;
         fprintf(stderr,"%c\n",c);
         ev[i] = c;
         i++;
     }
     ev[i] = '\0';

     fprintf(stderr,"%s\n",ev);

     i = 0;
     while ((c=fgetc(fp)) != ';'){
```

```
        if ((c == '\n')||(c == EOF)) {
           eoln = 1;
           break;
        }
        fprintf(stderr,"%c\n",c);
        val[i] = c;
        i++;
     }
     val[i] = '\0';

     fprintf(stderr,"%s\n",val);

     /* events and value read are placed in NADF record-like structure */

     if (j == TABLEN) return TCAPOVFL;

     event_table[j].length=52;
     event_table[j].tstp_id=1;
     event_table[j].tstp_lg=4;
     event_table[j].timestamp=tmstp;
     event_table[j].ev_id=2;
     event_table[j].ev_lg=16;
     strncpy(event_table[j].event,ev,16);
     event_table[j].val_id=3;
     event_table[j].val_lg=16;
     strncpy(event_table[j].value,val,16);
     j++;
    }

    etlen = j;
    if (c == EOF) return VEOF;

    /* reading finished, table has been correctly */

    return OK;
}


/* The main procedure opens a KAOS trace file and converts its records into NADF
     records.
 * A new, non-existing, file is created by the procedure and will contain the NADF
 * obtained trace.
 */
main(int argc, char **argv)
{
     int rc,i;
     FILE *fdin;
     int fdout;
     char *nadfrec;

     if (argc != 3) {
           fprintf(stderr, "%s: incorrect argument count\n", *argv);
           fprintf(stderr, "usage: %s native-file-name nadf-file-name\n",
     argv[0]);
           exit(1);
     }
     if ((fdin = fopen(argv[1],"r")) == NULL) {
           fprintf(stderr, "%s: cannot open file %s\n", argv[1]);
           exit(1);
     }
     if ((fdout = creat_NADF(argv[2])) < 0) {
           fprintf(stderr, "%s: cannot create: %s: %d\n", *argv, argv[2], fdout);
           exit(1);
     }
     while ((rc = kfread(fdin)) == OK) {
         i=0;
         while (i<etlen) {
           nadfrec = malloc(sizeof(struct nadf_rec));
```

```
            bcopy((char *) &(event_table[i].length), (char *) nadfrec,
                                                    sizeof(struct nadf_rec));
            if ((rc = write_NADF(fdout, nadfrec, 1)) != 0) {
                  fprintf(stderr, "%s: cannot write: %d\n", *argv, rc);
                  exit(1);
            }
            free(nadfrec);
            i++;
        }
    }
    if (rc != VEOF) {
        if (rc == TCAPOVFL) {
            fprintf(stderr, "%s: max event capacity reached for timestamp: %d\n",
                                                                  argv[0],
                                                            rc);
            exit(1);
        }
        fprintf(stderr, "%s: error reading file: %d\n", argv[1], rc);
        exit(1);
    }
    if ((rc = fclose(fdin)) != 0) {
            fprintf(stderr, "%s: cannot close file\n", argv[1]);
            exit(1);
    }
    if ((rc = close_NADF(fdout)) != 0) {
            fprintf(stderr, "%s: cannot close nadf file %s\n", *argv, argv[2]);
            exit(1);
    }
}
```

A.4. Implémentation du logiciel ORKA

Le code complet documenté est disponible sur :

www.info.fundp.ac.be/~sbrohez/ORKA

A.5. Obstacle Recognition with KAOS : ORKA, an implementation based on ASAX

Un exemplaire de l'article envoyé pour soumission au workshop RV'02 - Second Workshop on *Runtime Verification* (http://ase.arc.nasa.gov/rv2002/) est disponible sur :

www.info.fundp.ac.be/~sbrohez/ORKA

A.6. Code RUSSEL obtenu par l'utilisation de notre application pour l'illustration des exigences de « la mine »

Syntaxe des exigences introduites dans notre logiciel :

□ **[AND(/wdlevel > 600,dpfail = 'false') → ◊≤ 10 (alarm = 'true')]**

□ **[(/gdlevel > 430) → ◊≤ 10 (alarm = 'true')]**

□ **[AND(dpfail = 'false', /wdlevel >= 400) → o (/wdlevel < 400)]**

□ **[(/gdlevel > 430) → ◊≤ 10 (pump = 'false')]**

Code RUSSEL obtenu :

```
global internal activeRule1 : integer;
global internal activeRule2 : integer;
global internal activeRule3 : integer;
global internal activeRule4 : integer;


/* Les correspondances entre les evenements et leur signification
 *
 * ev0 = wdlevel (valeur integer)
 * ev1 = dpfail (valeur string)
 * ev2 = alarm (valeur string)
 * ev3 = gdlevel (valeur integer)
 * ev4 = alarm (valeur string)
 * ev5 = dpfail (valeur string)
 * ev6 = wdlevel (valeur integer)
 * ev7 = wdlevel (valeur integer)
 * ev8 = gdlevel (valeur integer)
 * ev9 = pump (valeur string)
 *
 */

rule Supervisor(tstp:integer;ev0:string;valeur0:string;ev1:string;valeur1:string;
          ev2:string;valeur2:string;ev3:string;valeur3:string;
          ev4:string;valeur4:string;ev5:string;valeur5:string;
          ev6:string;valeur6:string;ev7:string;valeur7:string;
          ev8:string;valeur8:string;ev9:string;valeur9:string);

var val0,val3,val6,val7,val8,val1,val2,val4,val5,val9:string;
begin
       val0 := valeur0;
       val1 := valeur1;
       val2 := valeur2;
       val3 := valeur3;
       val4 := valeur4;
       val5 := valeur5;
       val6 := valeur6;
       val7 := valeur7;
       val8 := valeur8;
       val9 := valeur9;

       if (tstp = strToInt(TIMESTP)) -->
              begin
                 if (match(EVENT,ev0)=1) --> begin   val0 := VALUE; val6 := VALUE;
                                                     val7 := VALUE end;
                    (match(EVENT,ev1)=1) --> begin val1 := VALUE; val5 := VALUE end;
                    (match(EVENT,ev2)=1) --> begin val2 := VALUE; val4 := VALUE end;
                    (match(EVENT,ev3)=1) --> begin val3 := VALUE; val8 := VALUE end;
                    (match(EVENT,ev9)=1) --> val9 := VALUE
```

```
                fi;
                trigger off for_next Supervisor(tstp,ev0,val0,ev1,val1,ev2,val2,
                                                ev3,val3,ev4,val4,ev5,val5,ev6,val6,
                                                ev7,val7,ev8,val8,ev9,val9)
            end;
         true -->
            if ((activeRule1 = 0) and ((bytesToInt(val0) > 600)
                and (match(val1,'false')=1)) ) -->
                begin
                   activeRule1 := 1;
                   trigger off for_current Rule1(tstp,'alarm',val2,10);
                   trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1,ev2,val2,
                                                      ev3,val3,ev4,val4,ev5,val5,ev6,val6,
                                                      ev7,val7,ev8,val8,ev9,val9)
                end;
                ((activeRule2 = 0) and (bytesToInt(val3) > 430 )) -->
                begin
                   activeRule2 := 1;
                   trigger off for_current Rule2(tstp,'alarm',val4,10);
                   trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1,ev2,val2,
                                                      ev3,val3,ev4,val4,ev5,val5,ev6,val6,
                                                      ev7,val7,ev8,val8,ev9,val9)
                end;
                ((activeRule3 = 0) and ((match(val5,'false')=1)
                and (bytesToInt(val6) >= 400))) -->
                begin
                   activeRule3 := 1;
                   trigger off for_current Rule3(tstp,'wdlevel',val7,1);
                   trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1,ev2,val2,
                                                      ev3,val3,ev4,val4,ev5,val5,ev6,val6,
                                                      ev7,val7,ev8,val8,ev9,val9)
                end;
                ((activeRule4 = 0) and (bytesToInt(val8) > 430 )) -->
                begin
                   activeRule4 := 1;
                   trigger off for_current Rule4(tstp,'pump',val9,10);
                   trigger off for_current Supervisor(tstp,ev0,val0,ev1,val1,ev2,val2,
                                                      ev3,val3,ev4,val4,ev5,val5,ev6,val6,
                                                      ev7,val7,ev8,val8,ev9,val9)
                end;
                true --> trigger off for_current Supervisor(tstp+1,ev0,val0,ev1,val1,
                                                      ev2,val2,ev3,val3,ev4,val4,ev5,val5,
                                                      ev6,val6,ev7,val7,ev8,val8,ev9,val9)
            fi
      fi
end;

rule Rule1(tstp:integer;ev0:string;valeur0:string;delay:integer);
/*Achieve[AlarmTriggered] on DrowningDanger*/
var val0:string;
begin
      val0 := valeur0;
      if (tstp = strToInt(TIMESTP)) -->
            begin
               if (match(EVENT,ev0)=1) --> val0 := VALUE
               fi;
               trigger off for_next Rule1(tstp,ev0,val0,delay)
            end;
         true -->
            if (delay = 0) -->
                  if (match(val0,'true')=1) -->
                     begin
                         activeRule1 :=0;
                         println('Rule for Requirement Achieve[AlarmTriggered] on
                                     DrowningDanger dies (',tstp,')')
                     end;
```

```
                        true -->
                        begin
                            println('Alarm ! - Obstacle detected in Requirement
                                Achieve[AlarmTriggered] on DrowningDanger (',tstp,')');
                            activeRule1 := 0
                        end
                    fi;
                true -->
                    if (match(val0,'true')=1) -->
                        begin
                            activeRule1 :=0;
                            println('Rule for Requirement Achieve[AlarmTriggered] on
                                        DrowningDanger dies (',tstp,')')
                        end;
                        true --> trigger off for_current Rule1(tstp + 1,ev0,val0,
                                                                delay - 1)
                    fi
            fi
        fi
end;

rule Rule2(tstp:integer;ev0:string;valeur0:string;delay:integer);
/*Achieve[AlarmTriggered] on Suffocation*/
var val0:string;
begin
        val0 := valeur0;
        if (tstp = strToInt(TIMESTP)) -->
            begin
                if (match(EVENT,ev0)=1) --> val0 := VALUE
                fi;
                trigger off for_next Rule2(tstp,ev0,val0,delay)
            end;
          true -->
            if (delay = 0) -->
                if (match(val0,'true')=1) -->
                    begin
                        activeRule2 :=0;
                        println('Rule for Requirement Achieve[AlarmTriggered] on
                                    Suffocation dies (',tstp,')')
                    end;
                    true -->
                    begin
                        println('Alarm ! - Obstacle detected in Requirement
                                Achieve[AlarmTriggered] on Suffocation (',tstp,')');
                        activeRule2 := 0
                    end
                fi;
                true -->
                    if (match(val0,'true')=1) -->
                        begin
                            activeRule2 :=0;
                            println('Rule for Requirement Achieve[AlarmTriggered] on
                                        Suffocation dies (',tstp,')')
                        end;
                        true --> trigger off for_current Rule2(tstp + 1,ev0,val0,
                                                                delay - 1)
                    fi
            fi
        fi
end;

rule Rule3(tstp:integer;ev0:string;valeur0:string;delay:integer);
/*Next[WaterLevelControlled]*/
var val0:string;
begin
        val0 := valeur0;
        if (tstp >= (strToInt(TIMESTP) - delay)) -->
```

```
            begin
               if (match(EVENT,ev0)=1) --> val0 := VALUE
               fi;
               trigger off for_next Rule3(tstp,ev0,val0,delay)
            end;
            true -->
               if (delay = 0) -->
                  if (bytesToInt(val0) < 400) -->
                     begin
                        activeRule3 :=0;
                        println('Rule for Requirement Next[WaterLevelControlled]
                                 dies (',tstp,')')
                     end;
                     true -->
                     begin
                        println('Alarm ! - Obstacle detected in Requirement
                                 Next[WaterLevelControlled] (',tstp,')');
                        activeRule3 := 0
                     end
                  fi;
                  true -->
                     if (bytesToInt(val0) < 400 ) -->
                        begin
                            activeRule3 :=0;
                            println('Rule for Requirement Next[WaterLevelControlled]
                                       dies (',tstp,')')
                        end;
                        true --> trigger off for_current Rule3(tstp + 1,ev0,val0,
                                                               delay - 1)
                     fi
               fi
         fi
end;

rule Rule4(tstp:integer;ev0:string;valeur0:string;delay:integer);
/*Achieve[PumpTurnedDown]*/
var val0:string;
begin
         val0 := valeur0;
         if (tstp = strToInt(TIMESTP)) -->
            begin
               if (match(EVENT,ev0)=1) --> val0 := VALUE
               fi;
               trigger off for_next Rule4(tstp,ev0,val0,delay)
            end;
            true -->
               if (delay = 0) -->
                  if (match(val0,'false')=1) -->
                     begin
                         activeRule4 :=0;
                         println('Rule for Requirement Achieve[PumpTurnedDown]
                                 dies (',tstp,')')
                     end;
                     true -->
                        begin
                           println('Alarm ! - Obstacle detected in Requirement
                                    Achieve[PumpTurnedDown] (',tstp,')');
                           activeRule4 := 0
                        end
                  fi;
                  true -->
                     if (match(val0,'false')=1) -->
                        begin
                            activeRule4 :=0;
                            println('Rule for Requirement Achieve[PumpTurnedDown]
                                       dies (',tstp,')')
                        end;
```

```
                    true --> trigger off for_current Rule4(tstp + 1,ev0,val0,
                                                           delay - 1)
                fi
            fi
        fi
end;

init_action;
        begin
           activeRule1 := 0;
           activeRule2 := 0;
           activeRule3 := 0;
           activeRule4 := 0;
           trigger off for_next Supervisor(0,'wdlevel','0','dpfail','defaultVal',
                           'alarm','defaultVal','gdlevel','0','alarm','defaultVal',
                           'dpfail','defaultVal','wdlevel','0','wdlevel','0',
                           'gdlevel','0','pump','defaultVal')
        end.
```

Trace utilisée (au format natif – un simple fichier texte) :

```
0;gdlevel&400;wdlevel&300;pump&true;dpfail&false;alarm&false;
                                    miners_digging&true;miners_inside&250
1;gdlevel&431
2;gdlevel&432
3;gdlevel&434
4;gdlevel&436
5;gdlevel&438;wdlevel&320
6;gdlevel&440
7;gdlevel&460
8;gdlevel&470
9;gdlevel&480;miners_digging&false
10;gdlevel&500;miners_digging&true
11;gdlevel&520;alarm&true;miners_inside&0;miners_digging&false
12;gdlevel&540;pump&false
13;gdlevel&530;wdlevel&310
14;gdlevel&520;wdlevel&300
15;gdlevel&510;wdlevel&290
16;gdlevel&500
17;gdlevel&490
18;gdlevel&480;wdlevel&270
19;gdlevel&470
20;gdlevel&460
21;gdlevel&450
22;gdlevel&440
23;gdlevel&425;alarm&false;miners_inside&250;miners_digging&true
24;gdlevel&420
25;gdlevel&410
26;gdlevel&400
27;wdlevel&250
28;miners_digging&false
29;miners_digging&true
30;wdlevel&230
31;wdlevel&235
32;wdlevel&230
33;wdlevel&220
34;wdlevel&240
35;wdlevel&260
36;wdlevel&280
37;wdlevel&300
38;wdlevel&320;miners_digging&false;miners_inside&230
39;wdlevel&340;miners_digging&true
40;wdlevel&360
41;wdlevel&380
42;wdlevel&420
43;wdlevel&440
```

```
44;wdlevel&480
45;wdlevel&520
46;wdlevel&560
47;wdlevel&600
48;wdlevel&640;miners_digging&false
49;wdlevel&680;miners_digging&true
50;wdlevel&720
51;wdlevel&760
52;wdlevel&800
53;wdlevel&840
54;wdlevel&860;alarm&true;miners_inside&0;miners_digging&false
55;wdlevel&800;pump&true
56;wdlevel&800
57;wdlevel&700
58;wdlevel&600
59;wdlevel&500;gdlevel&310
60;wdlevel&400
61;wdlevel&350
62;wdlevel&330
63;wdlevel&310
64;wdlevel&300
65;alarm&false;miners_inside&250;miners_digging&true
66;wdlevel&380
67;wdlevel&460
68;wdlevel&540;miners_digging&false;miners_inside&250
69;wdlevel&620;gdlevel&350;miners_digging&true
70;wdlevel&700;gdlevel&390
71;wdlevel&780;gdlevel&420
72;wdlevel&860;gdlevel&450;alarm&true;miners_inside&0;
                                                  miners_digging&false
73;gdlevel&460;wdlevel&940;pump&false
74;gdlevel&470;wdlevel&1020
75;gdlevel&480;wdlevel&1100
76;gdlevel&470;wdlevel&1180
77;gdlevel&460;wdlevel&1260
78;gdlevel&450;wdlevel&1340;miners_digging&false
79;gdlevel&440;wdlevel&1420;miners_digging&true
80;gdlevel&430;wdlevel&1600
81;wdlevel&1680;gdlevel&420
82;wdlevel&1580;gdlevel&410
83;wdlevel&1480;gdlevel&390
84;wdlevel&1380;gdlevel&380
85;wdlevel&1280;gdlevel&370
86;wdlevel&1180;gdlevel&360
87;wdlevel&1080
88;wdlevel&980;gdlevel&350
89;wdlevel&880;gdlevel&340
90;wdlevel&780;gdlevel&330
91;wdlevel&680;gdlevel&320
92;wdlevel&580;alarm&false;miners_inside&250;miners_digging&true
93;wdlevel&480;gdlevel&310
94;wdlevel&380
95;wdlevel&330
96;wdlevel&320
97;wdlevel&310
98;wdlevel&305
99;wdlevel&300;gdlevel&300;miners_digging&false;miners_inside&250
100;miners_digging&true
```

Résultat obtenu (avec les « morts » des règles indiquées) :

```
begin parsing description file ...
         asax :     code_bytes.asa
Processing audit trail ...

Alarm ! - Obstacle detected in Requirement Achieve[PumpTurnedDown] (11)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (11)
Rule for Requirement Achieve[PumpTurnedDown] dies (12)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (12)
Rule for Requirement Achieve[PumpTurnedDown] dies (13)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (13)
Rule for Requirement Achieve[PumpTurnedDown] dies (14)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (14)
Rule for Requirement Achieve[PumpTurnedDown] dies (15)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (15)
Rule for Requirement Achieve[PumpTurnedDown] dies (16)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (16)
Rule for Requirement Achieve[PumpTurnedDown] dies (17)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (17)
Rule for Requirement Achieve[PumpTurnedDown] dies (18)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (18)
Rule for Requirement Achieve[PumpTurnedDown] dies (19)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (19)
Rule for Requirement Achieve[PumpTurnedDown] dies (20)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (20)
Rule for Requirement Achieve[PumpTurnedDown] dies (21)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (21)
Rule for Requirement Achieve[PumpTurnedDown] dies (22)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (22)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (43)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (44)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (45)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (46)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (47)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (48)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (50)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (51)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (52)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (53)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (54)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (54)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (55)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (56)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (56)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (57)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (57)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (58)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (59)
Rule for Requirement Next[WaterLevelControlled] dies (60)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (68)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (69)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (70)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (72)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (72)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (72)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (72)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (73)
Rule for Requirement Achieve[PumpTurnedDown] dies (73)
Rule for Requirement Achieve[PumpTurnedDown] dies (73)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (73)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (73)
Rule for Requirement Achieve[PumpTurnedDown] dies (74)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (74)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (74)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (74)
Rule for Requirement Achieve[PumpTurnedDown] dies (75)
```

```
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (75)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (75)
Rule for Requirement Achieve[PumpTurnedDown] dies (76)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (76)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (76)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (76)
Rule for Requirement Achieve[PumpTurnedDown] dies (77)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (77)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (77)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (78)
Rule for Requirement Achieve[PumpTurnedDown] dies (78)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (78)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (78)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (79)
Rule for Requirement Achieve[PumpTurnedDown] dies (79)
Rule for Requirement Achieve[AlarmTriggered] on Suffocation dies (79)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (79)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (80)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (80)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (81)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (82)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (82)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (83)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (84)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (84)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (85)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (86)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (86)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (87)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (88)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (88)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (89)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (90)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (90)
Rule for Requirement Achieve[AlarmTriggered] on DrowningDanger dies (91)
Alarm ! - Obstacle detected in Requirement Next[WaterLevelControlled] (92)
Rule for Requirement Next[WaterLevelControlled] dies (93)

end of audit trail reached.
Processing completion rules ...

End of emulation.

user time div HZ : 0.010000
system time div HZ: 0.010000
total time div HZ : 0.020000
```